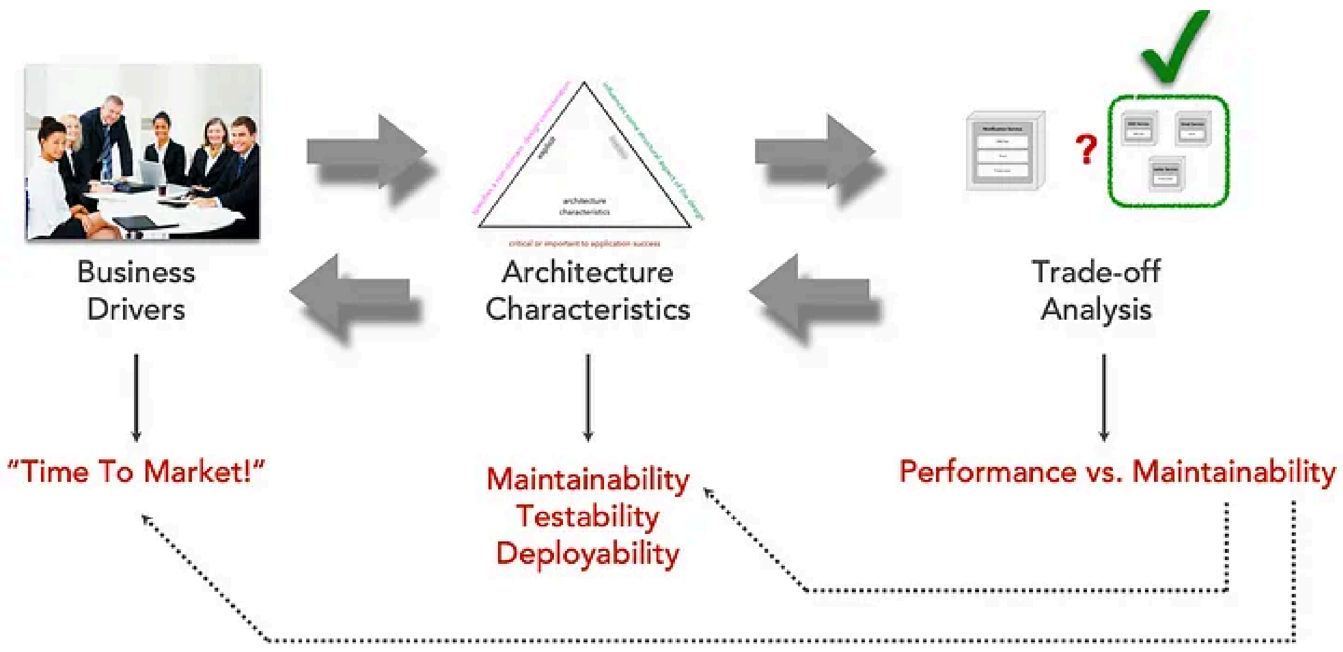
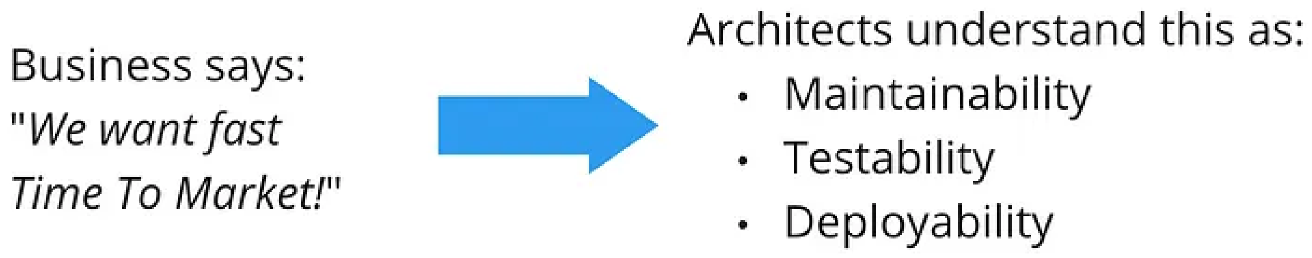
Modern System Architecture. Qualitative and quantitative analysis. Study case for analyzing Architecture tradeoffs.

Igor Asomadinov · Follow
16 min read · Apr 7, 2023

How do you make that decision, and how do you know if it's the most appropriate given all possible solutions in system architecture? Let's consider the following use case. It all starts with the business drivers and listening to the business, and finding out their goals.



For example, let's say the business says. *We need extremely fast time to market*. In other words, we must get our changes and features out to our customers. And our bug fixes as fast as possible. So what we do, as architects, we translate that into architectural characteristics. For example, *time to market* equates to three main architectural characteristics, high levels of *maintainability*, the ability to find and locate and change our code, *testability* which is not only the ease of but the completeness of testing, and finally, *deployability*, which is about the frequency of deployment. At the overall risk when we deploy our software and the ceremony, in other words, the time spent deploying that. So this now forms the context, the basis of our trade-off analysis. Now we have the decision to make, and we analyze the trade-offs.

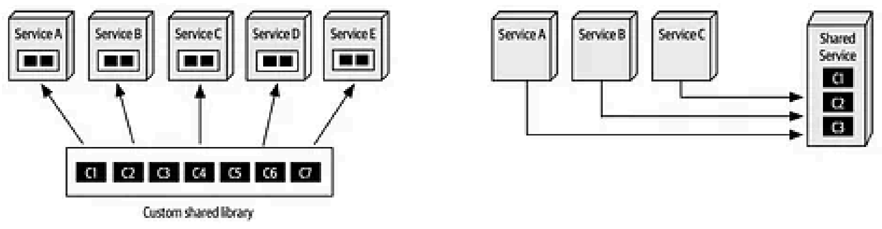

Business says:
"We want fast Time To Market!"
Architects understand this as:
• Maintainability
• Testability
• Deployability

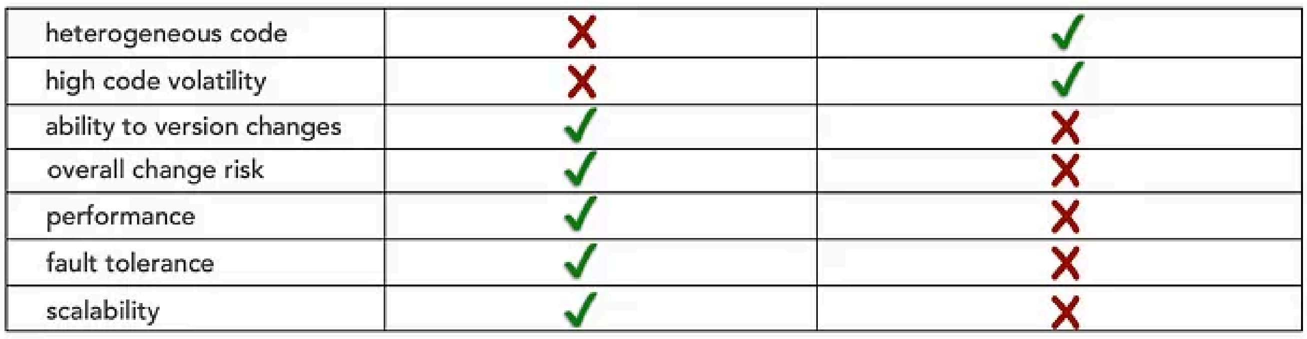
Let's boil down to a tradeoff of performance versus maintainability. Those are the two main trade-offs for a particular decision. How do we make a choice? What we do, is we tie it back to the architecture characteristics and then also tie it back to the business drivers and find out that while we. As developers and architects are technologists, they get pumped up about performance. We find out that maintainability is the more important characteristic, and that's how we do that tradeoff analysis. So let's look at some trade-off analysis and some tips and techniques.

And here is our first modern tradeoff analysis tip.

Tip #1: Watch out for the "out of context" trap when analyzing trade-offs.

The very first out of the starting gate. So I can't decide whether to use *shared libraries* or *shared services* for all the expected functionality in my microservices ecosystem or my distributed architecture. I mean, these are two of the most popular choices. **I don't know how to make this choice.** Let's start analyzing these. So what about this kind of criteria?





heterogeneous code	✗	✓
high code volatility	✗	✓
ability to version changes	✓	✗
overall change risk	✓	✗
performance	✓	✗
fault tolerance	✓	✗
scalability	✓	✗

From the book "The Fundamentals of Software Architecture" by Mark Richards, Neal Ford

Heterogeneous code? Well, if we take a look at those two. A shared library performs much better. I'm going to have to write that as a shared library for each one of the programming languages. But we also see a shared library losing in high code volatility. We have a high code volatility in that custom Jar Library. We're going to be continuous, having a lot of churns, retesting and redeploying all of our services, using those not with the shared service because we simply write that shared service in whatever language, and that's the only service we change or deploy. However, what about the ability diversion changes? Now we start to see the pendulums switching over because it's pretty straightforward to version. Library and manage those versions, or it's not so much with the shared service. Also, shared services have a lot more risk because they're real-time and change once I deploy that shared service. All the other services are now using it, so we have a bit more control and a little less risk with a shared library because I converted it and made that backward-compatible. Now from an operational perspective, performance, fault tolerance, and scalability. All so much more effective with the custom shared library. Because think about it, with the shared service performance, I've got additional latency, network latency, and possible database latency. I also have possible security latency. Fault tolerance if that shared service goes down. All my services that need access are now nonoperational, not so with a shared library. Because it's compiled, but it's compile-bound. And also, from a scalability perspective, if everybody needs that shared service, I will have to scale that out to meet a man of all the services using it. So if we add up this scorecard, you can see the clear best practice here, can't you? And that is the custom **shared library** wins hands down, which is why it's so popular.

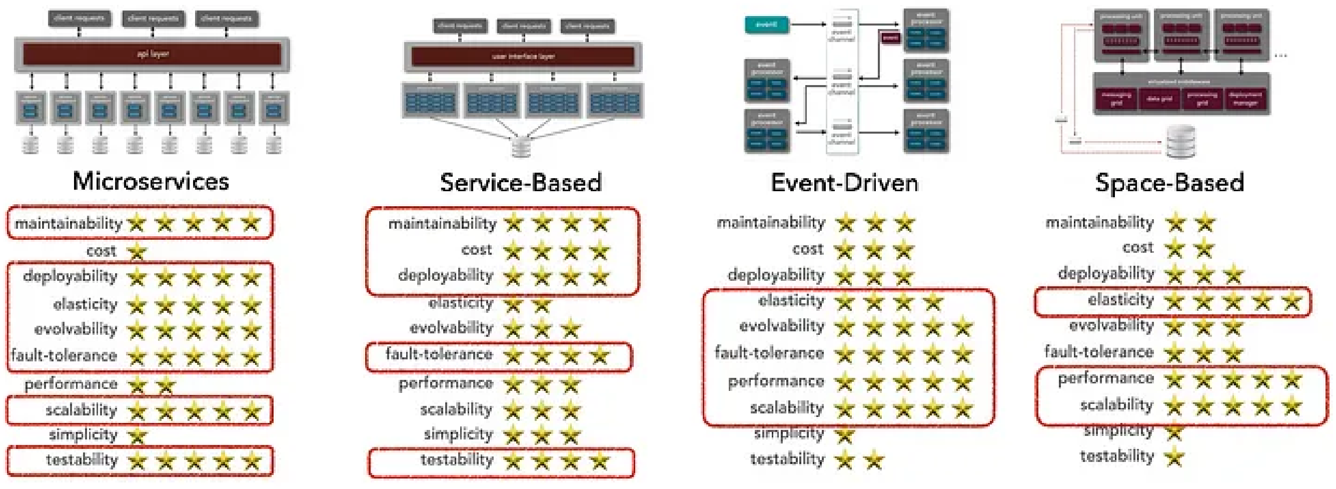
Now, this is an effective way of teasing out the corresponding tradeoffs. But the out-of-context trap occurs when we stop here and score what is better for performance and fault tolerance. We're not so concerned about those. What we're really worried about is all the change that frequently occurs in this shared functionality are heterogeneous code and high code volatility.

And as a matter of fact, you can see what wins now — that is "**Shared service.**" And so this is kind of the 1st and most basic one. Please. Don't fall into this trap.

Tip #2: "Use qualitative analysis to iterate on the design, leading to quantitative analysis."

To iterate on your design. Consider these examples of what I mean between qualitative and quantitative analysis.

Qualitative analysis. It analyzes the quality of several things. At the same time, *quantitative research* measures metrics between several things. Well, what happens if we don't have metrics yet? **How do I know which one to choose?** Because I don't have anything in existence yet. That's where *qualitative analysis* fits in. Let me show you how that works. So we're going to take all of these architectural styles.



From the book "The Fundamentals of Software Architecture" by Mark Richards, Neal Ford

And we will leverage the star ratings from the book "The Fundamentals of Software Architecture". And here are those star ratings.

Microservices are great for maintainability, elasticity, fault tolerance, scalability, and testability.

Service based? Pretty solid on maintainability, good in areas of that maintainability. Cost, fault tolerance.

Event-driven? Ohh, it's a superpower, folks. In terms of performance, you're up performance, scalability, elasticity, and evolvability.

Space-based architecture that leverages tuple space. It's the sweet spot in high elasticity, performance, and scalability levels.

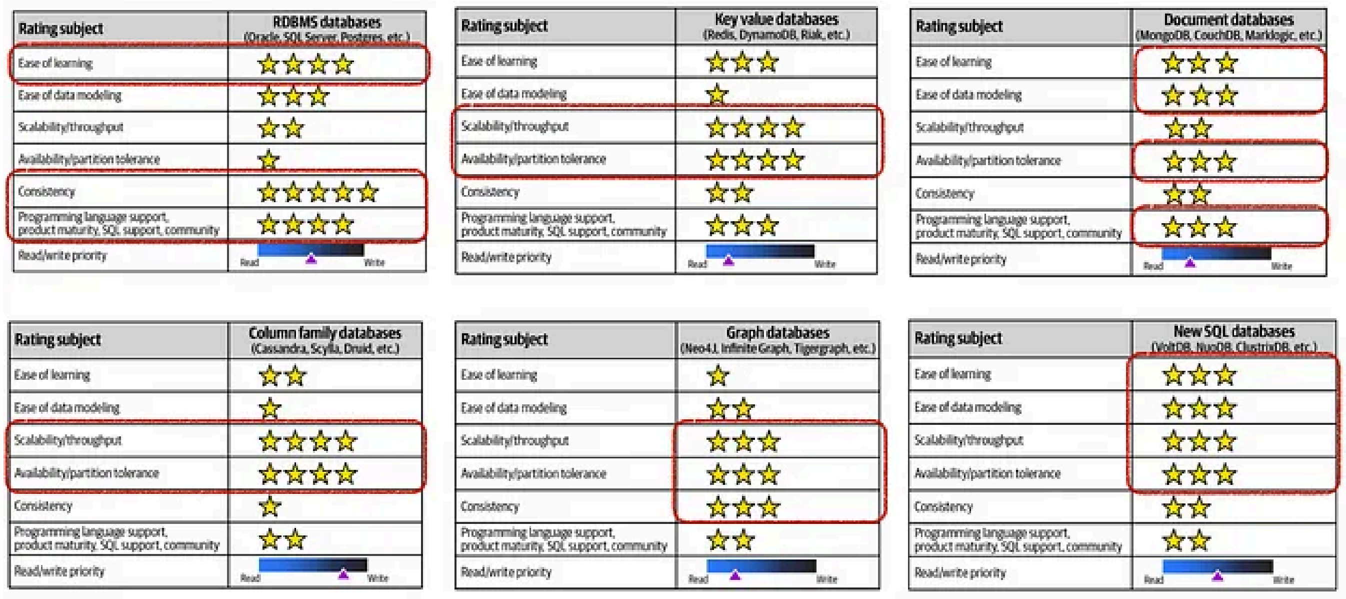
So we can take a look at all of these superpowers. But wait. Let's go back to that first tip. What is our context? Do you remember the *Time to Market*? It is the most important thing to the business. That translates to *maintainability, testability, and deployability*. All these have great superpowers, but let's put a context on it. Let's show you qualitative analysis combined with the out-of-context trap. So this, folks, is what we're concerned about. Make tenability, deployability, and testability, i.e., maintainability, deployability, and testability.

Now I can apply qualitative analysis comparing the quality. Of two things, see which one supports that characteristic better. Even though we don't have exact metrics yet, we know we need fast time to market, so we take microservices and compare them against service based. We know supporting time to market is the most critical aspect of this system, but I will go one step further because what are the tradeoffs in choosing microservices? Cost, performance. And simplicity is pretty bad, meaning it's an incredibly complex architectural style. We get great time to market at the sacrifice of cost, performance, and simplicity. Ouch. Our systems would be a little slower, and it will be super expensive.

But going back to our analysis. We take a look at service-based. Cost — Four stars. And look at performance — three stars, which ain't bad, and simplicity — 3 stars right here. So you see, we're only losing 1 star on each important thing. But we can increase other things where the tradeoffs start to form a bit more of a balance between every characteristic because we're doing qualitative analysis.

The bottom line may be *service based*. It might be a better approach because, you see, we did the tradeoff analysis, we did it qualitatively, but we took one step further. To actually see what those trade-offs were all about.

Next — Data. RDBMS relational database management systems. So I can't decide which database types would be most appropriate for the new system. You know, so many times. We default to relational databases when in fact, we have a lot of choices. Let's do a tradeoff analysis on databases.



Rating subject	RDBMS databases (Oracle, SQL Server, Postgres, etc.)
Ease of learning	★★★★
Ease of data modeling	★★★
Scalability/throughput	★★
Availability/partition tolerance	★
Consistency	★★★★★
Programming language support, product maturity, SQL support, community	★★★★
Read/write priority	Read Write

Rating subject	Key value databases (Redis, DynamoDB, Riak, etc.)
Ease of learning	★★★
Ease of data modeling	★
Scalability/throughput	★★★★
Availability/partition tolerance	★★★★
Consistency	★★
Programming language support, product maturity, SQL support, community	★★★
Read/write priority	Read Write

Rating subject	Document databases (MongoDB, CouchDB, Marklogic, etc.)
Ease of learning	★★★
Ease of data modeling	★★★
Scalability/throughput	★★
Availability/partition tolerance	★★★
Consistency	★★
Programming language support, product maturity, SQL support, community	★★★
Read/write priority	Read Write

Rating subject	Column family databases (Cassandra, Scylla, Druid, etc.)
Ease of learning	★★
Ease of data modeling	★
Scalability/throughput	★★★★
Availability/partition tolerance	★★★★
Consistency	★
Programming language support, product maturity, SQL support, community	★★
Read/write priority	Read Write

Rating subject	Graph databases (Neo4J, Infinite Graph, Tigergraph, etc.)
Ease of learning	★
Ease of data modeling	★★
Scalability/throughput	★★★
Availability/partition tolerance	★★★
Consistency	★★★
Programming language support, product maturity, SQL support, community	★★
Read/write priority	Read Write

Rating subject	New SQL databases (VoltDB, NuoDB, ClustrixDB, etc.)
Ease of learning	★★★
Ease of data modeling	★★★
Scalability/throughput	★★★
Availability/partition tolerance	★★★
Consistency	★★
Programming language support, product maturity, SQL support, community	★★
Read/write priority	Read Write

From the book "The Fundamentals of Software Architecture" by Mark Richards, Neal Ford

Let's start with relational database management systems. And these star ratings came from the book "The Fundamentals of Software Architecture". as well.

We can see that relational databases' superpowers are ease of learning. Consistency is five stars here, and the amount of support, community support, and language support is perfect. That's its sweet spot.

But what about other database types? How about a key-value database? This would be like Dynamo DB, Redis. What are its superpowers? Well, it's really about scalability, throughput, and partition tolerance.

What about document databases? It doesn't have any five stars. But, it's pretty average across the board, pretty balanced on ease of learning, data modeling, availability, and the programming, which supports. We lose some consistency, but we gain a little bit more of that language-supported availability.

Column databases, things like Cassandra. Sweet spot — raw throughput, scalability, and availability. But not good for many other things, especially ease of data modeling.

How about things like Neo4J or Tiger Graph? Things like graph databases look. Only a few superpowers, but really good at reading, scalability, throughput, availability, and consistency. Pretty well balanced with great read support, but you'll be slower.

New SQL: This is kind of forming a balance between NoSQL databases and relational databases. It fits right in the middle. Voltdb is a really common example of these. If we take a look here a pretty well balanced in terms of the ease of learning data modeling, scalability, and throughput i

Back to our use case: *Time to Market* — high levels of *maintainability, testability, and deployability*. Now that we come back here, we don't see those directly stated, which is very common. So we have to make a little bit of interpretation. We can start looking at our trade-offs.

The relational database supports that reasonable time to market, especially with the ease of learning and data modeling.

Key value. Now, just then, the ease of data modeling alone and the ease of learning. It will take longer to model new features to make changes.

But let's take a look at document databases. Pretty well-balanced. That would be another good consideration. Now you might get excited about things like Cassandra, but as a columnar database, it's complex. And the ease of learning, ease of data modeling and programming, language support, and entity support is much lower than others.

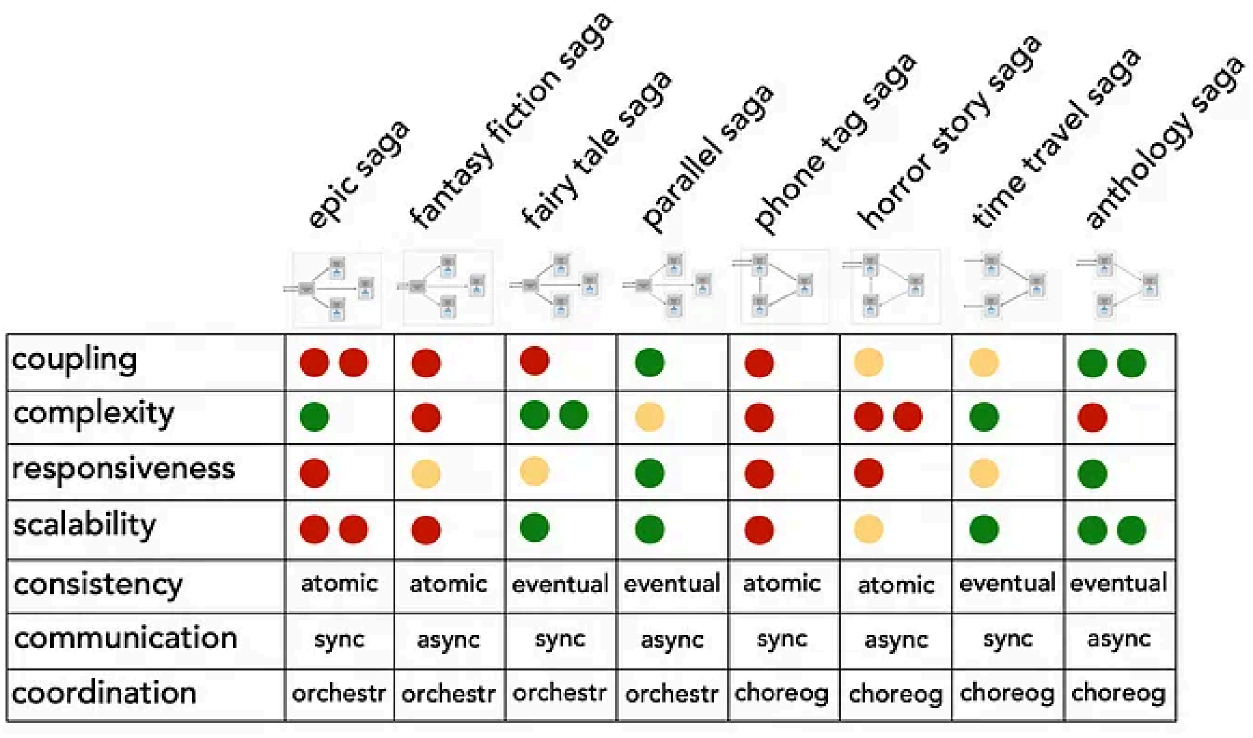
Graph databases — Nope, it's too hard to learn and model.

New SQL is pretty well balanced.

And so with this analysis, making this modern trade-off analysis, we could see three really good candidates: *New SQL vs Document database vs RDBMS*. And now, we can apply the type of data that we have to overlay this kind of analysis. To say, well, our data is not really relational in nature. It's more about really kind of a balance between documents and sort of wanting to keep SQL. Maybe New SQL would be a good option.

And so that's another excellent example of this qualitative analysis. Notice we're not using any numerical analysis here, as we don't have it yet.

Let's do one more, and then I will show you a quantitative analysis.



	epic saga	fantasy fiction saga	fairy tale saga	parallel saga	phone tag saga	horror story saga	time travel saga	anthology saga
coupling	🔴🔴	🔴	🔴	🟢	🔴	🟡	🟡	🟢🟢
complexity	🟢	🔴	🟢🟢	🟡	🔴	🔴🔴	🟢	🔴
responsiveness	🔴	🟡	🟡	🟢	🔴	🔴	🟡	🟢
scalability	🔴🔴	🔴	🟢	🟢	🔴	🟡	🟢	🟢🟢
consistency	atomic	atomic	eventual	eventual	atomic	atomic	eventual	eventual
communication	sync	async	sync	async	sync	async	sync	async
coordination	orchestr	orchestr	orchestr	orchestr	choreog	choreog	choreog	choreog

From the book "The Fundamentals of Software Architecture" by Mark Richards, Neal Ford

Let's start from: **I'm unsure which transactional saga I should use**. If we look at our context, time to market basically translates into getting changes out. As fast as possible. This is just the quality of something. And looking at this list, it really is all about complexity. Generally, the more complex, the harder it is to maintain, change, and deploy.

You can also ask why you would include and use something like an epic saga. It seems as if it is losing everywhere. It's about trade-offs. In some situations, you absolutely positively have to verify that each transaction happened correctly before moving on to the next step. And if an error occurs, you must go back and ensure all those things are cleaned up. You don't care how long it takes. If data consistency is the most important thing in architecture, and so in that case, you trade off performance and scalability and other things to get data above all.

Let's look at a fairy tale saga. It combines eventual consistency with synchronous communication using orchestration. It is the least complex solution we have.

Let's come back to that second step. We know supporting time to market. It's the most critical aspect. Now we've chosen a *fairy tale saga*. At least, I think that would be best. But what are the trade-offs you see so far? A very tangled tale saga produces high levels of coupling, and responsiveness really isn't that great. Now scalability is good. But the trade-off of that is coupling and responsiveness.

Maybe. There's another balance we should go and revisit. Let's do that. Do we have any other Greens in complexity? — Answer: *Parallel Saga*. We're adding more complexity, but it's still qualitatively good. And we get a little less coupling. Yet responsiveness stays about the same in comparison with the fairy tale saga. As a result, we can consider two options and apply a particular context to see what is most appropriate: *fairy tale saga* or p*arallel saga.*

simplicity
coupling
responsiveness

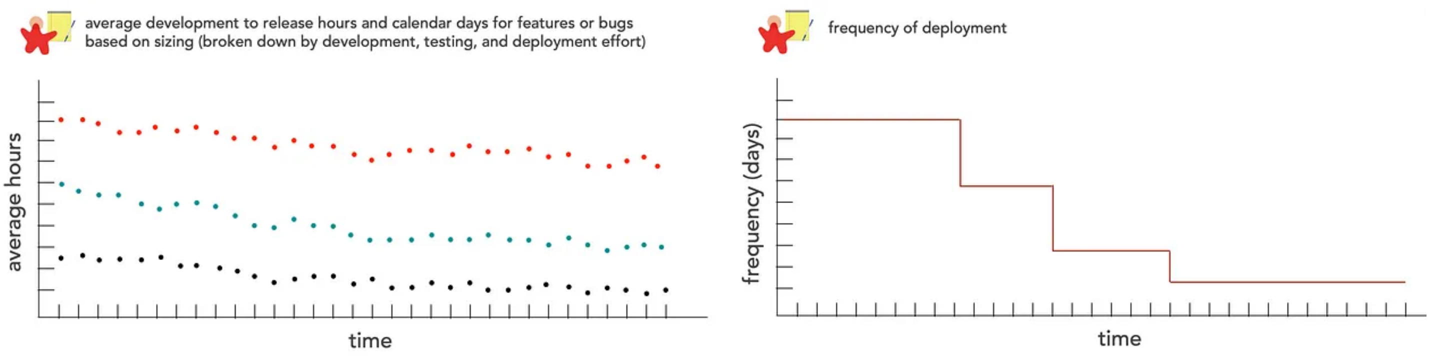
Quantitative Analysis

Now that the system is in place. Do we make the right choices? Because we applied qualitative analysis comparing the quality of things. But we need to demonstrate overall agility to support our fast time-to-market needs, and we can actually leverage architectural fitness functions. But time to market is kind of this little subjective thing. However, we can still measure it. And let me show you a bunch of ways to support quantitative analysis.

Here's what we can measure. We could record. Manually or something like JIRA through plugins in its automated workflow. We can then capture this information on the average *development to release hours* and *calendar days for any feature or bug* based on its sizing. You could make *story points. You* could do small, medium, large, and extra large. And the fact is we could measure and track this. Is it where we need to be? This is a quantitative analysis to fine-tune our design. So that's why qualitative analysis allows us to iterate, and then quantitative analysis will enable us to verify and refactor.

You want to make your average hours for small, medium, and large tickets go down. It's a matter of fact part of the time to market is testability. Can we measure something with testability? Aside from just the amount of time it takes? Yes, because testability is also about the completeness of testing. Are we shortcutting our testing cycles? To get faster time to market. I could *record the number of errors and bugs* in our system over time, and this should go down or at least remain somewhat consistent.

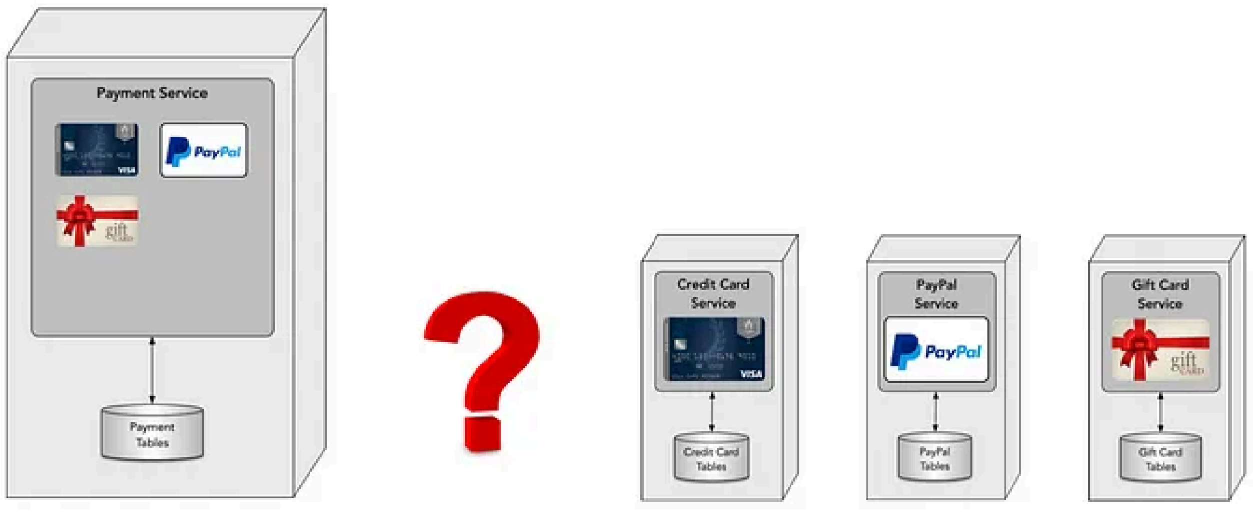
OK, we could record and actually measure something. And that has to do with deployability. What's the frequency of our deployments? Is it getting better or worse? I can measure this and hopefully see that we are deploying more frequently and faster. All right. So that is qualitative and quantitative analysis. Actually, those stubs should be up if the time to market is a big concern and the days between the frequencies could be shorter. Fixes new features out to our customers as fast as possible.



Model relevant business use-case scenarios to identify trade-offs.

I've been showing you many tips about how to do that trade-off analysis. But how do you determine, What the trade-offs are? So let's use an example of service granularity.

Here's a common question: **I wonder if we should have a single payment service or separate services**, one for a credit card, one for PayPal, one for Venmo, and one for gift cards.



All these kinds of different payments. We can apply modern tradeoff analysis to give us the most appropriate answer. So let's model some use cases.

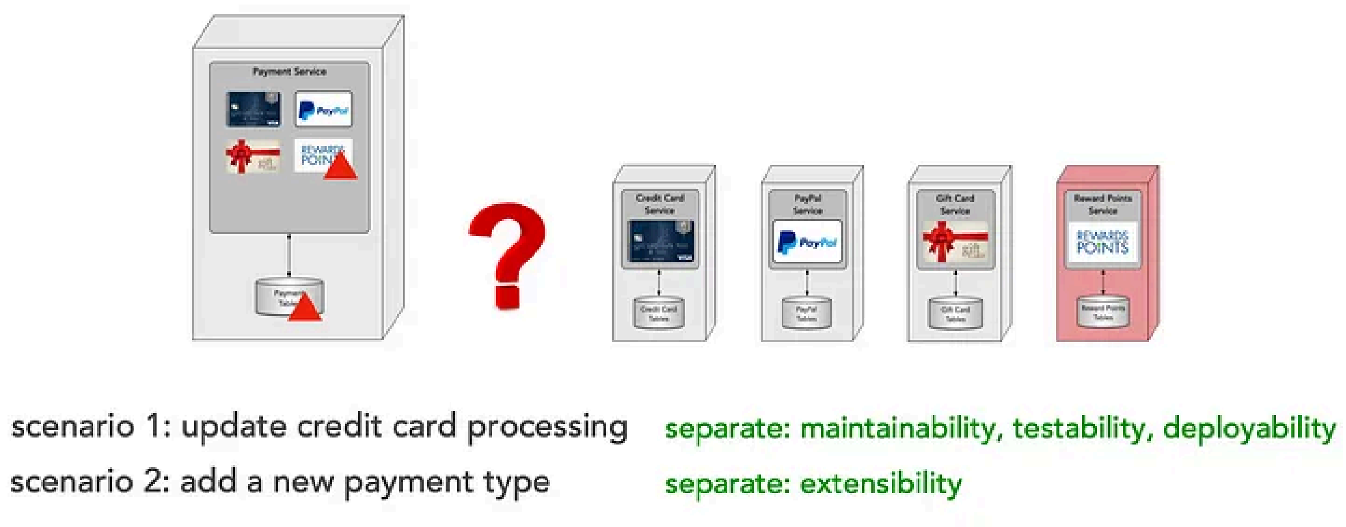
Scenario #1. If it's a single service, I must apply updates. I have to apply changes to the database in a single service. I've got larger testing. Hence a little bit less completeness of testing. My deployment scope is bigger. And I've got a little bit more deployment risk.

However, what if I break apart the services? Now, this is the only service I'm changing. Less testing scope, faster testing, complete testing. I'm not impacting PayPal or gift cards at all. , And so we see for updates. These give us: Maintainability, testability, and deployability.

But if we return to our first rule or the first law of software architecture, everything is a tradeoff. **If you think you found something that doesn't have a trade-off. Keep looking, Keep digging!!!** Because all it means is you haven't found it yet. Keep digging.

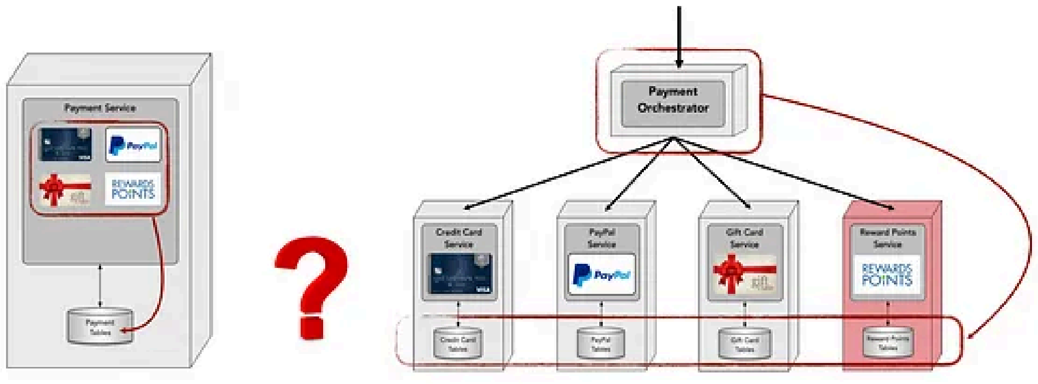
Scenario #2. We are going to be adding a lot of new payment types. We have plans for World Pay, Venmo, and Rewards points. If I add reward points to my single payment service. Guess what? I've got the same problems I did with updates. I've got a larger testing scope. I have to modify the database. I have to make changes to that service. I've got more deployment risk. It's going to take slower than that's going to take longer.

However, what if I separate those out? I create the database or the tables, and I just deploy them? I'm not impacting any other types of payment we have. So what we see is adding a new payment type gives us architectural extensibility, which relates to time to market.



Scenario #3. Customers can use multiple payment types for a single order. Well, what happens in the single service? Nothing. But we have a driver method. We write one single method. We've got a database transaction to do bits and rollbacks. Suppose one of those payment types doesn't work. We've got acid transactions.

What happens with separate payment types here? Do I have to create an orchestrator? A separate service and I have to reroute every endpoint from my API to go to that orchestrator, whether it's just a credit, card or credit card and PayPal. And, of course, I will have some performance issues here. There's more latency, and as a matter of fact, guess where my transactional unit of work is everybody. Setting the orchestrator means I've got a distributed transaction with eventual consistency. No more commits and rollbacks. Possibly going to have data integrity issues. Now we see the trade-off.



scenario 1: update credit card processing	separate: maintainability, testability, deployability
scenario 2: add a new payment type	separate: extensibility
scenario 3: use multiple types for payment	separate: performance, data consistency

We found the tradeoff, performance, and data consistency if we go with the Microservices solution. We return to business and ask, but what’s our biggest concern? And it is *Time to market*, i.e., maintainability, testability, and deployability. So we choose this. Knowing the analysis of our trade-off again. Which is performance and data consistency. Those become knowns now, unfortunately. We can’t really look at an intermediary, but we know those are concerns that maybe we can do something about.

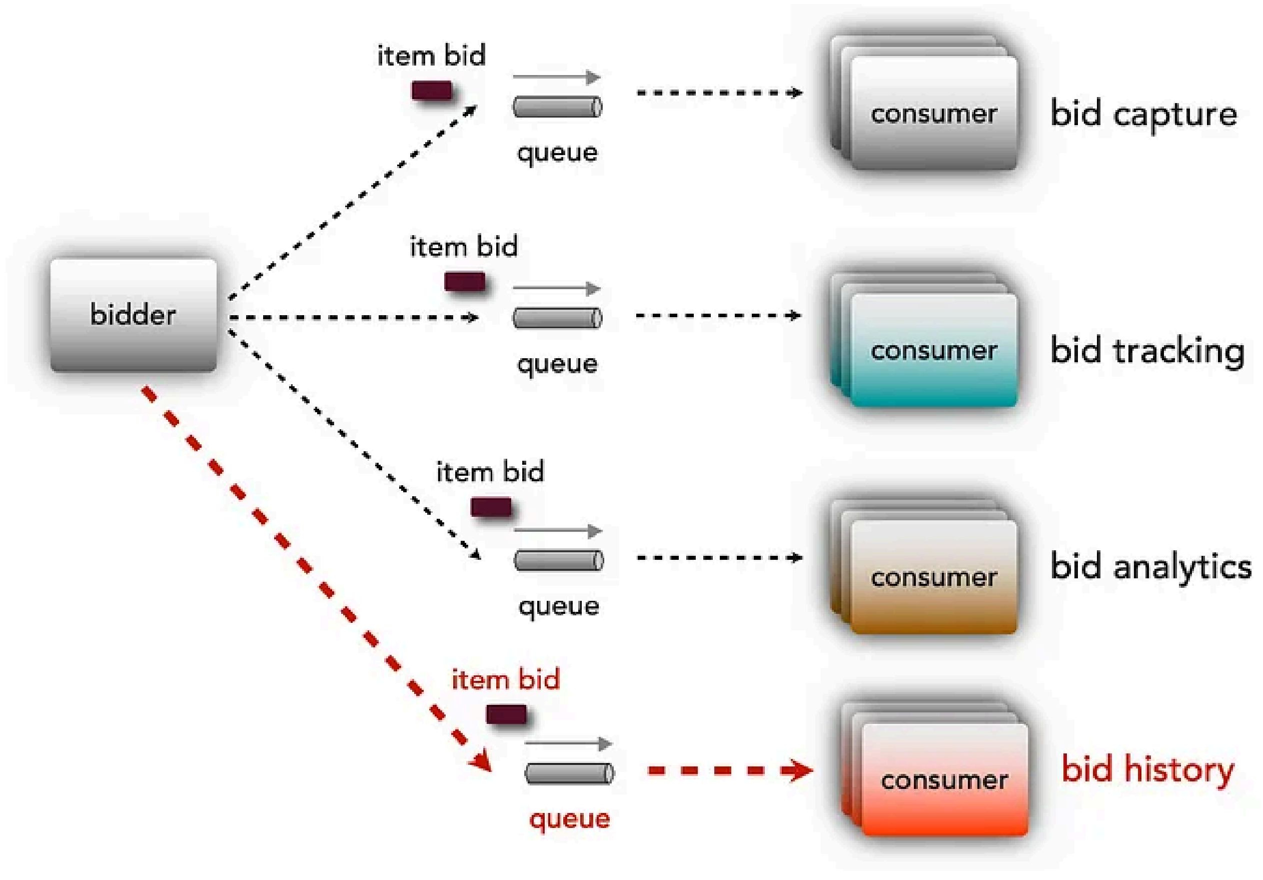
maintainability
testability
deployability
extensibility
performance
data consistency

And that is this last tip:

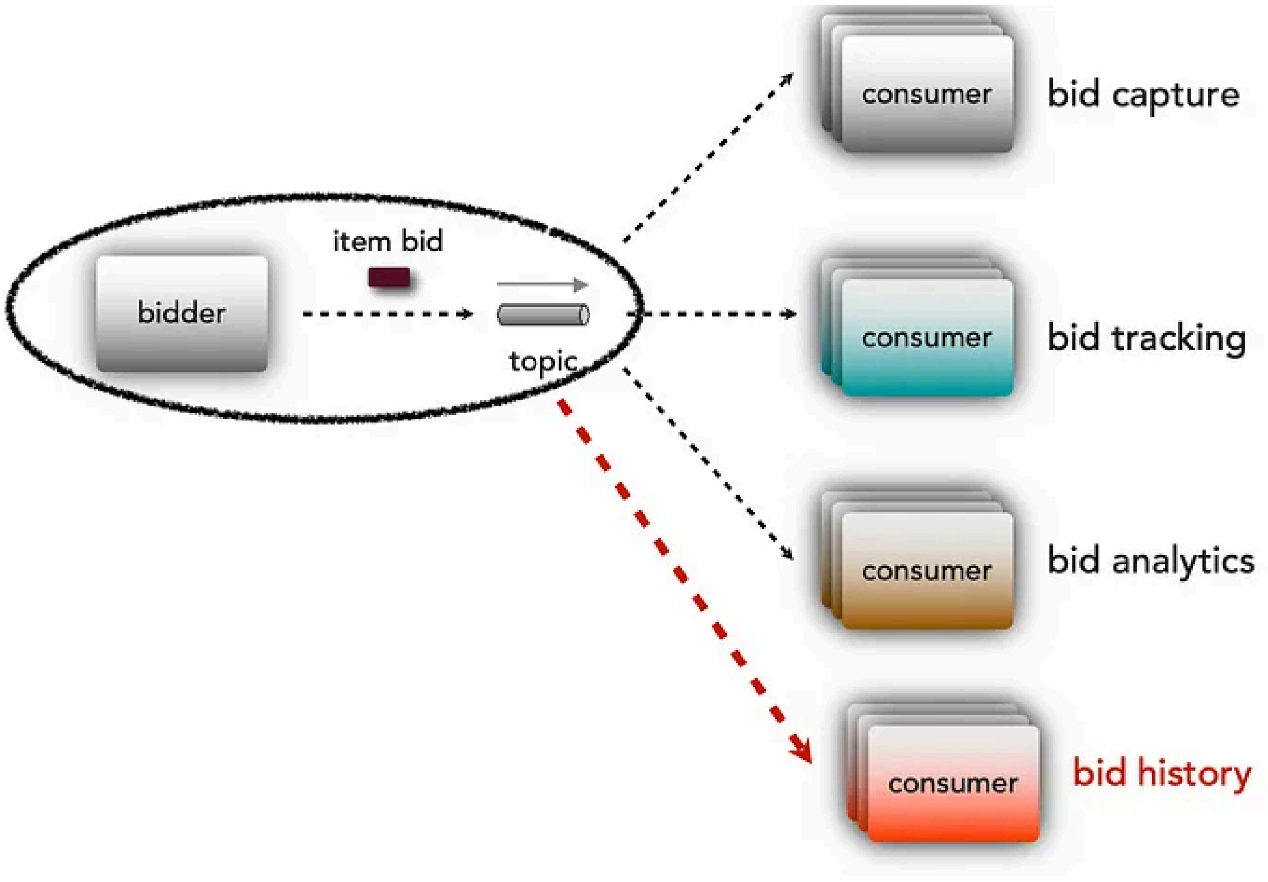
Tip #3. Avoid over-evangelizing a particular solution or technology. Any particular or given solution, technology, framework, product, or anything you try to avoid in modern trade-off analysis.

Example: Messaging systems. And you know, this usual opinion: “*Forget point-to-point queues. You should always use publish-and-subscribe broadcast messaging using* *topics because it supports architectural extensibility and evolutionary architecture — it's the greatest thing ever!"*

Let's say that we have an auction. We've got bidders and consumers that capture the bidding track that bid for a particular item. Someone will tell you: This is ridiculous if you're making a point to point. I have to have a queue for each consumer, and I can have a separate contract, the bidder. Service must know to go to three different brokers or queues. What if I wanted to add bid history? I want to track the history of bids. I'd have to create a new queue. I have to make a new contract. I have to modify that producer and bidder to a new point.



And if we use topics. If I wanted to add a new service, it would be effortless.

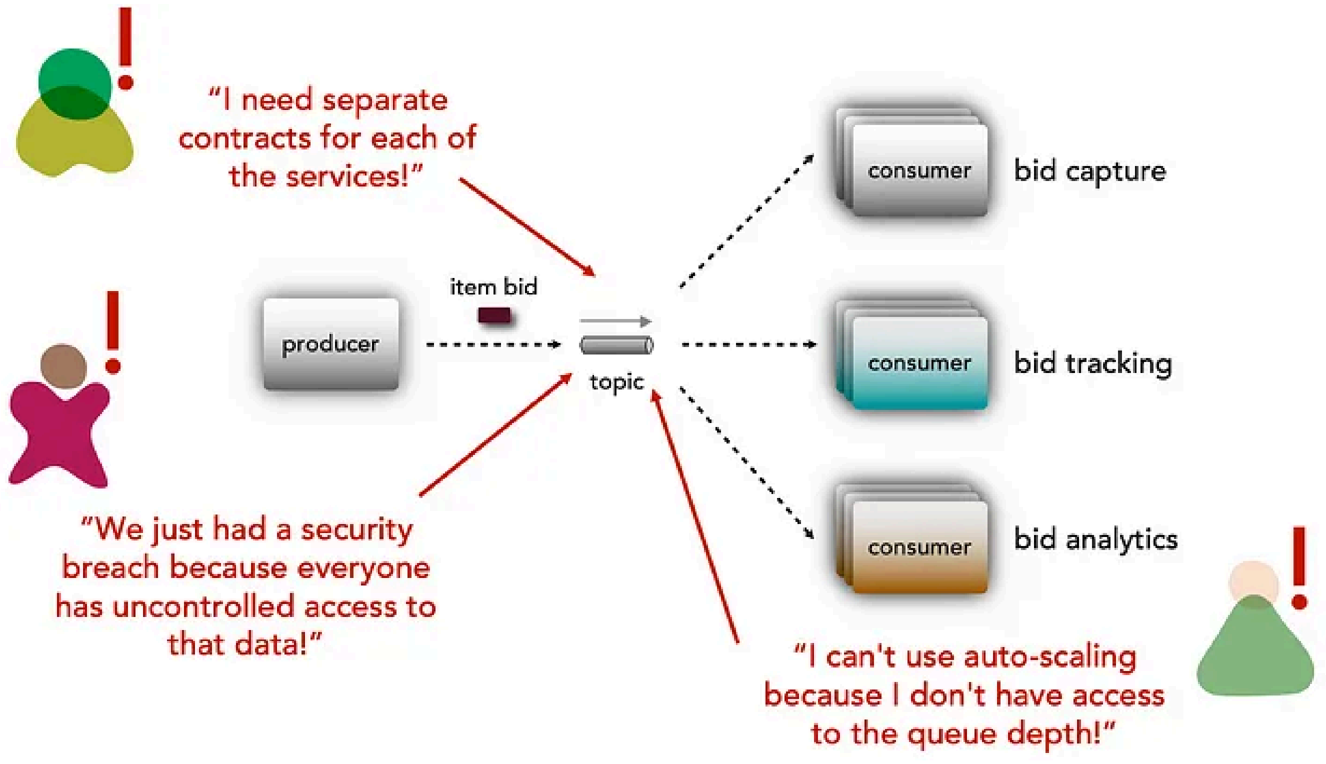


Meanwhile. A little bit later. What happens?

“*Uh oh hey, we got to have separate contracts. I can’t create a separate contract for each of these.*”

Another person says: “*Hey, we just had a security breach because everybody can access that data. And we actually had a hacker come in and subscribe to that.*“

And then somebody else said: “*We have a need here for auto-scaling. I need a queue depth for programmatic auto-scaling*”.


"I need separate contracts for each of the services!"
consumer
bid capture
item bid
producer
topic
consumer
bid tracking
"We just had a security breach because everyone has uncontrolled access to that data!"
consumer
bid analytics
"I can't use auto-scaling because I don't have access to the queue depth!"

Remember everybody. **Everything in software architecture is a trade-off**, and by evangelizing. You steer everybody away from identifying those trade-offs. We get all excited and excited about a particular trend. A solution you came up with. Please, you can get excited but don't over-evangelize it. Always look for the next shoe to drop because it is there.

Conclusion:

This article explored the concept of trade-offs in software architecture and how they can impact the development process. We discussed three key tips for managing trade-offs effectively:

1. Identify the underlying problem and the goals you want to achieve.
2. Consider multiple solutions and weigh the pros and cons of each.
3. Avoid over-evangelizing a particular solution or technology.

By keeping these tips in mind, developers can make informed decisions about the best approach for their project and avoid being too narrowly focused on a single solution. Remember, everything in software architecture is a trade-off, and it's essential to approach these trade-offs with a balanced and open-minded perspective.

Software Architecture System Design Concepts Business Requirements

Software Analysis

Follow

Written by Igor Asomadinov

219 Followers · 1 Following

Responses (1)

What are your thoughts?

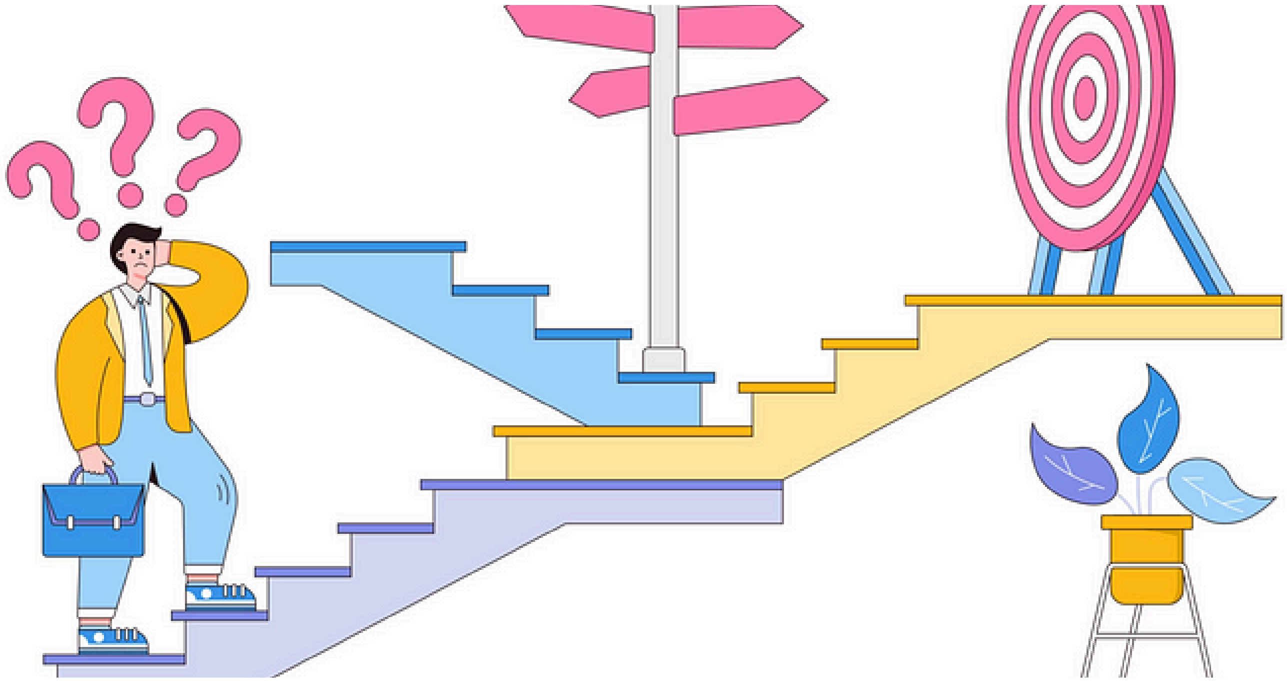
Respond

Sergiy Yevtushenko
Jun 15, 2023

Picture which you're using for #2 is very biased towards microservices and overall looks like it contains random values. For example: giving 5 stars for fault tolerance to system which even has no means to determine if all elements are up and… more

5 Reply

More from Igor Asomadinov

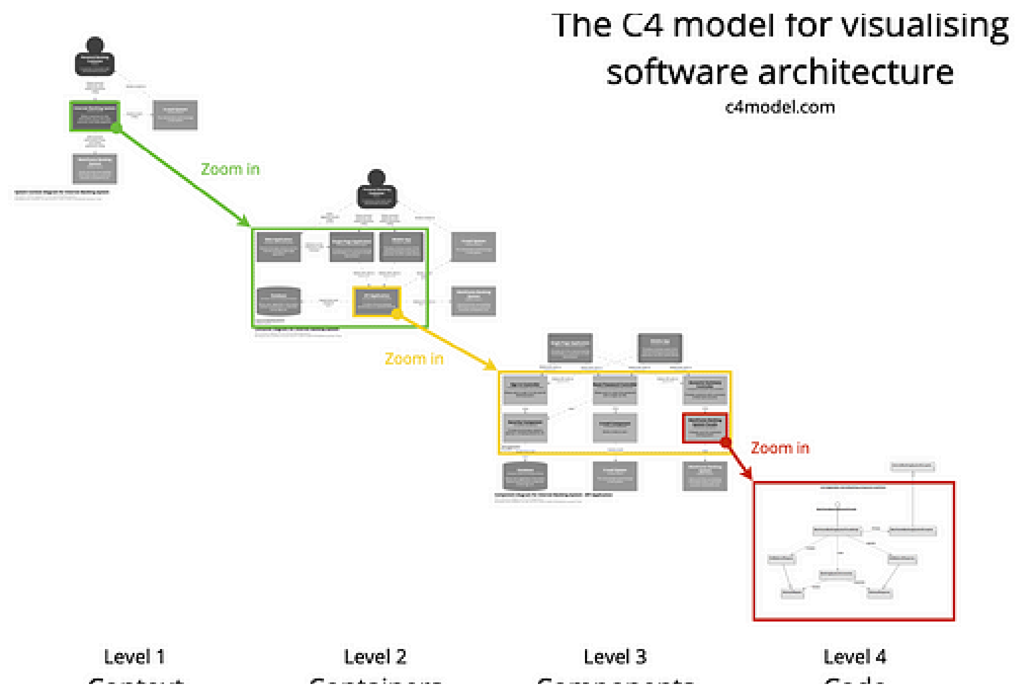


Igor Asomadinov

How to advance as Software Engineer. How to be a more Senior Software Engineer

Simple but not easy points on: “How to find ways that we can have more impact on the jobs that we’re doing”...

Apr 10, 2023 1K 16



Igor Asomadinov

Way of visualizing system architecture—C4 model technique

“Do Care” during planning and system creation. Separate pieces and details whenever you are showing components …

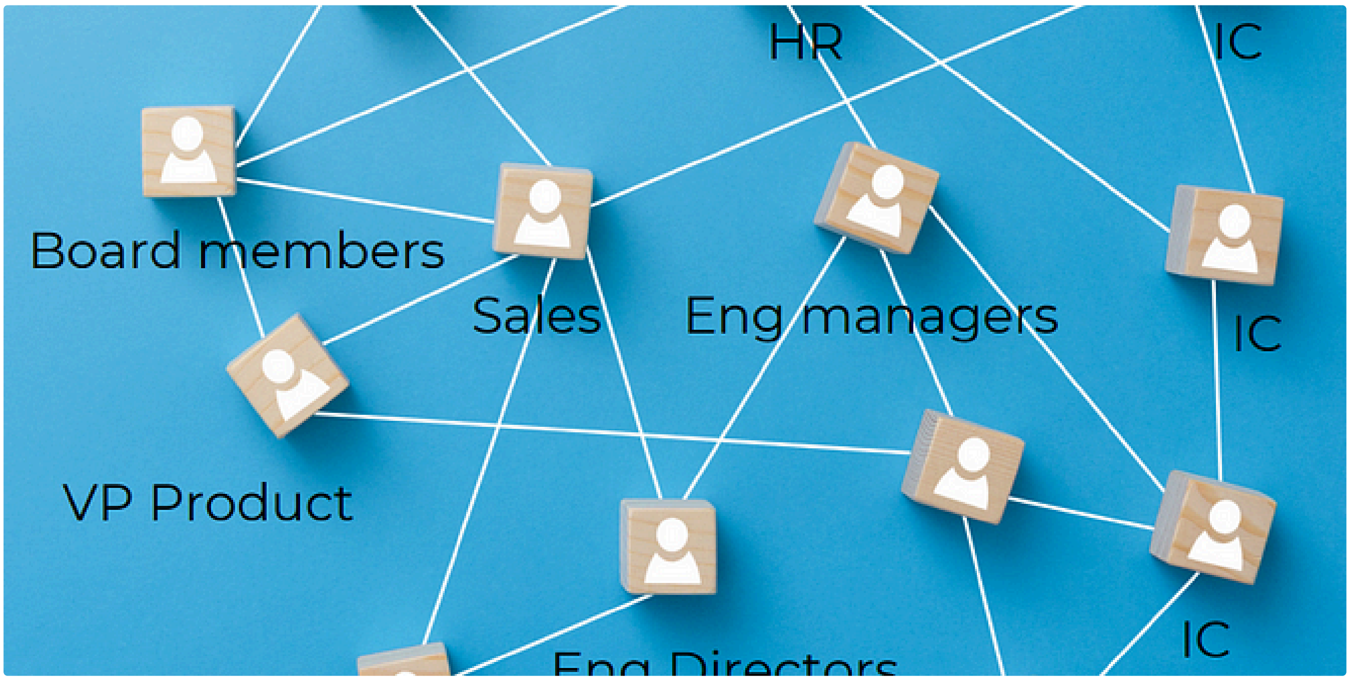
Dec 3, 2023 7

Igor Asomadinov

Dev guide for novice SW joining the project (edition for: CI/CD / Jira / Local Dev/ GitLab / Slack)

Eternal onboard that should help newcomers

May 18, 2024



Igor Asomadinov

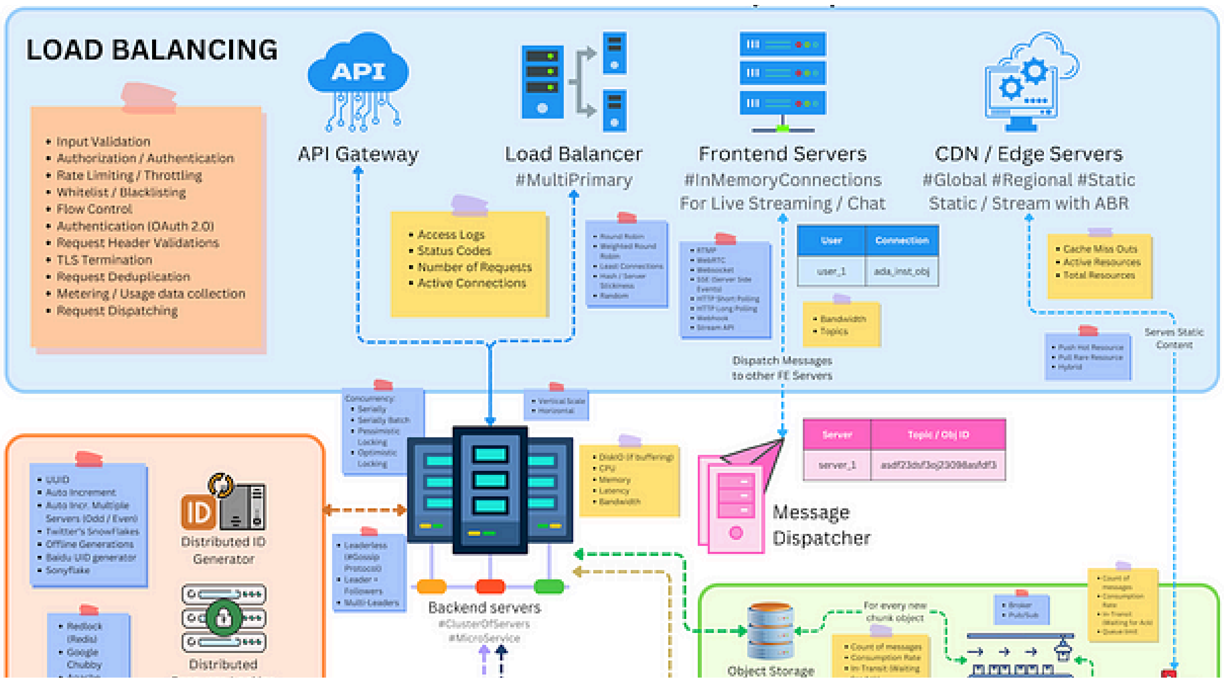
Being a senior engineering leader. Skills that you can start building already

What do you need to be a successful software engineering leader? Start working on the skills that will be helpful now and in the future.

Jun 3, 2023 84

See all from Igor Asomadinov

Recommended from Medium

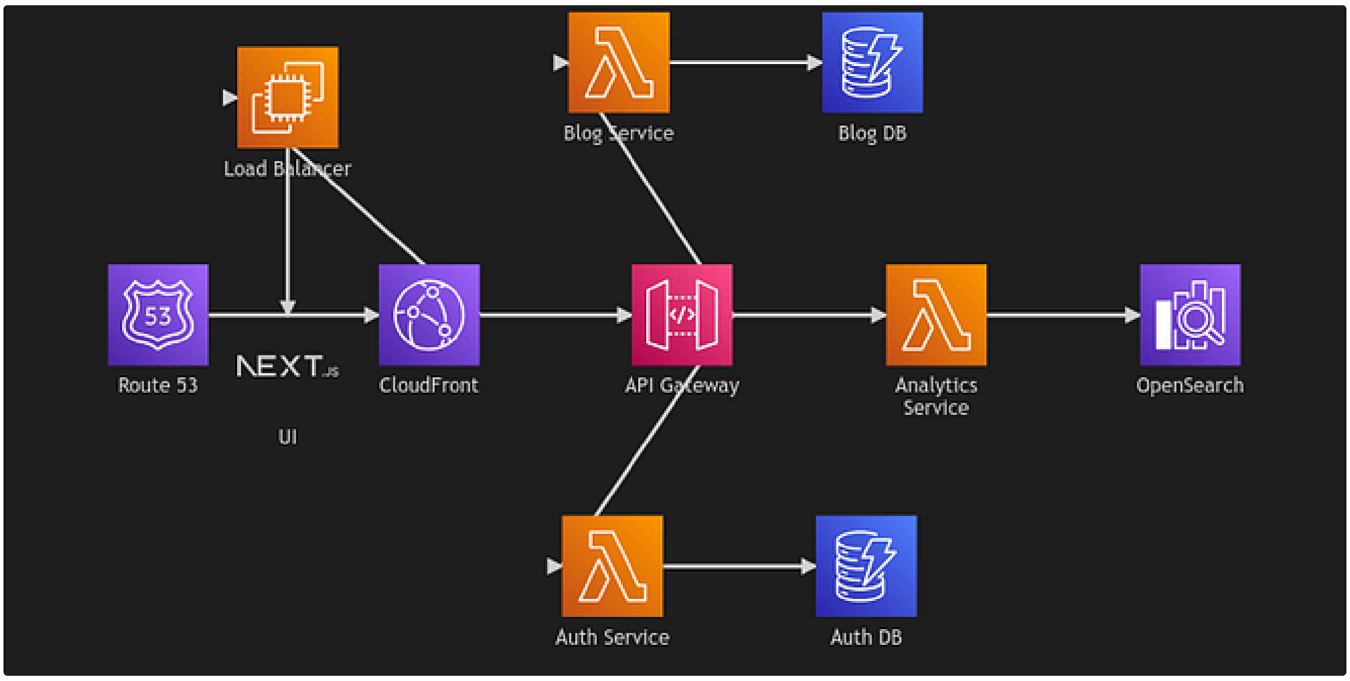


In ByteByteGo System Design Alliance by Love Sharma

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the…

Sep 18, 2023 9K 57



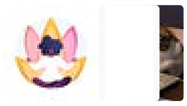
Kevin O'Shea

Architecture diagrams as code: Mermaid vs Architecture as Code

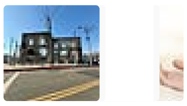
For many years I have been using Miro to visually document software, as well as to collaborate with my team. It is a fantastic tool, the…

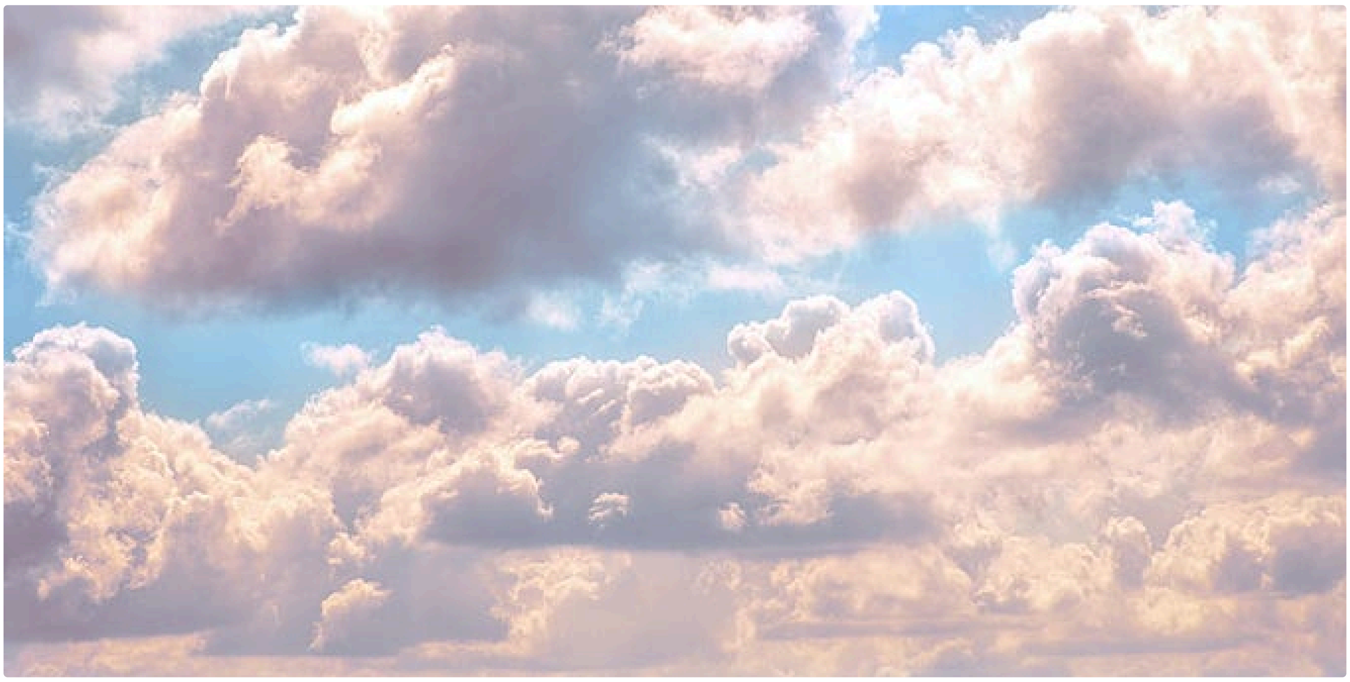
Nov 13, 2024 158 10

Lists


Stories to Help You Grow as a Software Developer
19 stories · 1592 saves

General Coding Knowledge
20 stories · 1906 saves


Staff picks
809 stories · 1617 saves



Ali Zeynalli

10 Must-Know Cloud Native Architecture Patterns

Sidecar/Sidekick, Ambassador, Scatter/Gather, BFF, Anti-Corruption Layer, CQRS, Event Sourcing, Service Mesh, Dumb-Smart Components…

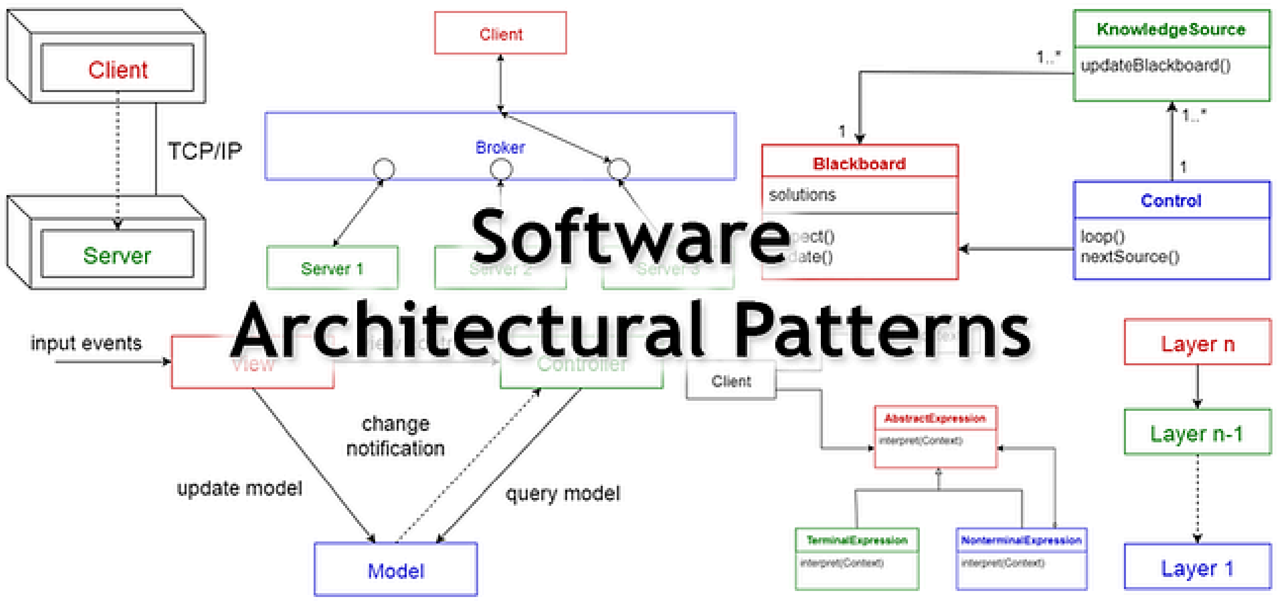
Jan 18 13

Talha Şahin

High-Level System Architecture of Booking.com

Take an in-depth look at the possible high-level architecture of Booking.com.

Jan 11, 2024 6.1K 48



In TDS Archive by Vijini Mallawaarachchi

10 Common Software Architectural Patterns in a nutshell

Ever wondered how large enterprise scale systems are designed? Before major software development starts, we have to choose a suitable…

Sep 5, 2017 41K 140

In Coding Odyssey by Shivam Srivastava

JP Morgan Java Developer Interview

Java Lead Interview Experience

Jan 24 342 13

See more recommendations