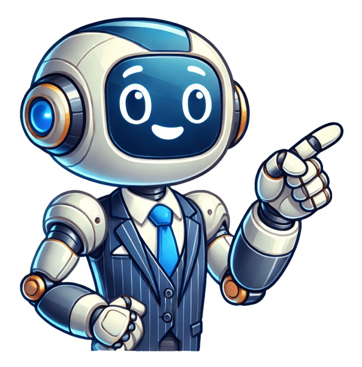
I'm not a robot



Book notes on "Monolith to Microservices" by Sam Newman These are my notes on Monolith to Microservices by Sam Newman.Practical and useful book on the subject.Key Insights Microservices are independently deployable services modeled around a business domain. Don’t focus on size but: How many microservices can you handle. How to define boundaries to get the most out of the microservices, without everything becoming a horrible coupled mess. Modular monolith still have the challenge of a monolith DB. DDD: Aggregate: self-contained unit that have a life cycle around them. Bounded context (BC) + Aggregate == unit of cohesion. The aggregate is a self-contained state machine that focuses on a single domain concept in our system, with the BC representing a collection of associated aggregates, with an explicit interface to the wider world. Microservices are not the goal. What are you going to achieve that you cannot with your current architecture? Have you considered alternatives to using microservices? How will you know if the transition is working? Reuse it not an outcome. When not to use microservices: Unclear domain. Startups. Customer installed and managed software. Dr John Kotter’s eight-step process for implementing org change (from Leading Change): Establish a sense of urgency. Creating the Guiding Coalition. Developing a Vision and Strategy. Communicating the change vision. Empowering employees for broad-based action. Generating short-team wins. Consolidating gains and producing more change. Anchoring new approaches in the culture. Small changes, little steps. The biggest the bet and bigger the accompanying fanfare, the harder is to pull out when it is going wrong. Pattern: Change Data Capture (CDC): Use when there is no other option. Pattern: Database-as-a-Service interface. When you are relying on network analysis to determine who is using your database, you are in trouble. Split the DB first or the code? Static reference data: Duplicate, dedicated schema, static library. Data service: only when creating microservices is cheap. BPM tools: issue is that they are non-dev tools that end up being used by devs. The more coupling, the earlier the pains will manifest: 2-10 services: Breaking changes. Reporting. 10-50: Ownership at scale. Developer experience. Running too many things. 50+: Loads of teams. Global vs local optimization. Orphaned services. In all: Robustness and resilience. Monitoring and troubleshooting. End-to-end testing. Solutions: No cross team tests. Consumer-driven contracts. Use automated release remediation and progressive delivery in addition to end-to-end tests. Without strong code ownership (one and only one team can change service, other teams can do pull requests to propose changes) a microservices’ architecture will grow into a distributed monolith. Microservices are independently deployable services modeled around a business domain. Start with the technology that you know. Don’t focus on size but: How many microservices can you handle. How to define boundaries to get the most out of the microservices, without everything becoming a horrible coupled mess. Modular monolith still have the challenge of a monolith DB. Other monoliths: Distributed monolith. Third-party black-box systems, both on premise and SaaS. Coupling types: Types. Implementation coupling. Temporal coupling (in the sense of sync calls). Deployment coupling (release trains). Domain coupling. DDD: Aggregate: self-contained unit that have a life cycle around them. Bounded context (BC): Hide implementation and internal details. BC + Aggregate == unit of cohesion. The Aggregate is a self-contained state machine that focuses on a single domain concept in our system, with the BC representing a collection of associated aggregates, again with an explicit interface to the wider world. Start by targeting services that encompass entire BC. You can split them further latter, hiding this decision. Microservices are not the goal. Key questions: What are you going to achieve that you cannot with your current architecture? Have you considered alternatives to using microservices? How will you know if the transition is working? Reuse it not an outcome. Limit the scope of expected outcomes. Possible whys: Improve team autonomy, alternatives: Modular monoliths. Assign responsibilities based on functional grounds. Self-servicing. Reduce time to market, alternatives: Value stream mapping to identify bottlenecks. Scale cost-effectively for load, alternatives: Vertical or horizontal scaling. Improve robustness: Being able to react to expected variations. Alternatives: Running multiple copies of your monolith. Use more reliable SW/HW. Automate manual processes. Scale number of developers, alternatives: Modular monolith (but less). Embrace new technology, alternatives: When not to use microservices: Unclear domain: Getting services wrong can be expensive due to large number of cross-service changes and overly coupled components. Startups. Customer installed and managed software: Due to increased operational complexity. Dr John Kotter’s eight-step process for implementing org change (from Leading Change): Establish a sense of urgency. Creating the Guiding Coalition: Try to bring somebody from “the business”. Developing a Vision and Strategy: Vision: realistic yet aspirational. Commitment to vision is important, but overly commitment to strategy can be dangerous Communicating the change vision: Face to face + broadcast. Empowering employees for broad-based action: Bandwidth change: increase capacity or reduce load. Generating short-team wins. Consolidating gains and producing more change: Anchoring new approaches in the culture: Communicate successes and failures. Small changes, little steps. Whiteboard is where the cost of change and the cost of mistake is the lowest. Where to start? Identify BC and their relationships: Just enough DDD to get started. Maybe use Event Storming Relationship show how easy/difficult should be to extract that BC (warn: code may disagree). Plot BC in: Is the transition working? Regular checkpoints. Agenda: Restate what you are trying to achieve. Does it still make sense? Review quantitative metrics. Ask for qualitative feedback: Happier in BVSSH. Decide if any change is needed. Avoid the skunk cost fallacy: The biggest the bet and bigger the accompanying fanfare, the harder is to pull out when it is going wrong. You have more options if you can change the monolith. Best if you can copy (not move) existing code. Consider refactoring into modular monolith first. Pattern: Stranger Fig Application: Steps: Identify what to migrate. Copy to microservice. Reroute calls to new microservice. Safe to rollback routing. There must be a clear way to redirect the calls to the new service. In case of HTTP, a reverse HTTP proxy in front of the monolith. Consider Ngnix + Lua if need something custom. If custom logic is very complex (like changing protocol from SOAP to gRPC), consider: Avoid putting the logic in the proxy, as it is a shared service between teams. Implement it in the microservice. Use service mesh. In message systems, the proxy consumes the monolith queue and does a content-based routing to two queues: new monolith queue and microservices one. While migrating, avoid any functionality change. Pattern: UI Composition: UI to call new microservice. Widget or page level. Mobile apps are monoliths, unless you can make changes without resubmitting them. Micro-frontend. Pattern: Branch by Abstraction: Steps: Create abstraction for the functionality to be replaced. Change clients to use new abstraction. Create new implementation that uses the new microservice. Switch over the new implementation. Cleanup. Verify branch by abstraction: If call to new implementation fails, call the old implementation. Pattern: Parallel Run: In strangler and branch by abstraction, call both implementation and compare results. Can check also performance. N-version programming: Implement functionality in several different ways, and do a parallel run, choosing the “correct” (quorum) one. For fault tolerance and to avoid bugs. Verification techniques: Use spy in microservices to record what will do, but without doing it. When duplicated side-effects are not ok. Github scientist. Not trivial, use just when there is a high risk. Pattern: Decorating Collaborator: When cannot or don’t want to change monolith. Proxy works as a decorator that calls new microservice in addition to old monolith. Pattern: Change Data Capture (CDC): React to changes happened in the data store. Typical implementations: DB triggers: Transaction log poll. Batch delta copier: Process that on a regular schedule scans the DB to find what data has changed since last run. Use when there is no other option. Shared DB: It is ok with: Read-only static reference data that is stable and has a clear owner. Pattern: Database-as-a-Service interface. Pattern: Database view: Less coupling than shared DB. When you are relying on network analysis to determine who is using your database, you are in trouble. Useful only for read-only. Pattern: Database wrapping service: Move database dependencies to service dependencies. When is too hard pulling the schema apart. More flexible than DB view. Can take writes. Stepping stone, buys you time. Pattern: Database-as-a-Service interface: Create a specific and dedicated DB to be accessed externally. Mapping engine options: CDC. Batch process. Build from an event log. Transferring Ownership: Pattern: Aggregate exposing monolith: Monolith expose a proper aggregate API. When a newly extracted microservice still needs data owned by the monolith. Maybe a future microservice. Pattern: Change data ownership: When monolith still depends on newly extracted microservice data. Ideally monolith should call new service API: Copying the data back to the monolith DB as an alternative. Data synchronization: Pattern: Synchronize data in application: Steps: Bulk synchronize data: If monolith was kept online, implement CDC to copy data since snapshot created. Synchronize on write, read from old schema. Synchronize on write, read from new schema. Decommission old schema. Pattern: Tracer write: Same as synchronize data in app, but one table at a time (instead of the whole bundled context) using the new microservice API. Split the DB first or the code? DB first: Easy, little short-term benefit. When concerned about performance or data consistency. Pattern: repository per BC: First step to understand dependencies. Pattern: database per BC: Bet for future split. Recommend for greenfield. Code first: Most common. Short-term improvements. Pattern: Monolith as data access layer: Same pattern as “aggregate exposing monolith”. Pattern: Multischema storage: New microservice uses new schema for new functionality/data, old schema for existing data. DB and code at the same time: Schema separation examples: Pattern: Split table: When table is owned by 2 or more BC. Lose referential integrity. Will need to chose the owner of the data. Pattern: Move foreign-key relationship to code: Increase latency: from 1 join query to 1 select + n service calls. Data consistency, deletion options: Check with all services before deleting: More coupling, reverse dependency. Don’t use. Handle 404/410 gracefully in dependant service. Don’t allow deletion: Soft delete/tombstone record. Static reference data: Duplicate: As it changes infrequently maybe ok. For large volume of data. Background process to update it. Dedicated schema: No duplication, always up to date. Allows for cross-schema joins. Static library: Data service: When creating a new microservice is cheap. Maybe can emit change events. Avoid distributed transactions. Sagas: Model transactions as business processes. Specially for long lived transactions. Rollback with compensating transactions. Two implementations: Orchestrated: Central coordinator. Command and control. Pro: easier to understand. Cons: coupling. Warn: Anemic microservices: coordinator having too much logic that should be in microservices. Different services can play the coordinator role for different flows. BPM tools: issue is that they are non-dev tools that end up being used by devs. Camuda and Zeebe are targeted to microservices developers. Choreographed: Distributed responsibility. Trust but verify. Usually event based. Pro: decoupled. Cons: harder to understand whole flow. Use correlation Id to build/know the state of the saga: Service to read all events to show view. More detailed in “Building microservices” book. Based on anecdotal experience. The more coupling, the earlier the pains will manifest: 2-10 services: Breaking changes. Reporting. 10-50: Ownership at scale. Developer experience. Running too many things. 50+: Loads of teams. Global vs local optimization. Orphaned services. In all: Robustness and resilience. Monitoring and troubleshooting. End-to-end testing. Ownership at scale: Without strong code ownership (one and only one team can change service, other teams can do pull requests to propose changes) a microservices’ architecture will grow into a distributed monolith. Breaking changes: Avoiding accidental breaking changes: Explicit schema to avoid structural breakages. Protolock to prohibit incompatible changes. To avoid semantic breakages: testing. Make it hard to change a service contract: Make it obvious, no magic, or generate schemas from code. Non-accidental: Do not break, but accrete. Give consumers time to migrate: Run two versions of the service. Support old and new endpoints in the service. Be more relaxed if changes are within a team. Reporting: Build a reporting specific schema. Monitoring and Troubleshooting: Log aggregation: First thing to do when implementing microservices. Tracing: API GW or service mesh to generate the correlation ID. Test in production: Towards observability: Distributed Systems Observability. Local developer experience: How many services to run locally? Solutions: Stubs. Point to instance running elsewhere. One remote env per developer: Slow to deploy to test changes. Cost. Mix local/remote: Telepresence for Kubernetes. Azure’s cloud functions. Running too many things: Deployment, configuration and management of instances becomes more difficult. Solution: Kubernetes. Function-as-a-Service (preferred). End-to-end testing: Even slower and even more brittle. Solutions: No cross team tests. Consumer-driven contracts: Use automated release remediation and progressive delivery in addition to end-to-end tests. Global vs local optimization: Solving the same problem twice. Divergent tech stack. Solutions: Cross-cutting group to raise awareness/make tech decisions. Robustness and resilience: Orphaned Services: In-house service registries that combines service discovery + code repository data. Did you enjoy it? Follow @DanLebrero or share! Tagged in : Architecture book notes How do you detangle a monolithic system and migrate it to a microservice architecture? How do you do it while maintaining business-as-usual? As a companion to Sam Newman’s extremely popular Building Microservices, this new book details a proven method for transitioning an existing monolithic system to a microservice architecture.With many illustrative examples, insightful migration patterns, and a bevy of practical advice to transition your monolith enterprise into a microservice operation, this practical guide covers multiple scenarios and strategies for a successful migration, from initial planning all the way through application and database decomposition. You’ll learn several tried and tested patterns and techniques that you can use as you migrate your existing architecture.Ideal for organizations looking to transition to microservices, rather than rebuildHelps companies determine whether to migrate, when to migrate, and where to beginAddresses communication, integration, and the migration of legacy systemsDiscusses multiple migration patterns and where they applyProvides database migration examples, along with synchronization strategiesExplores application decomposition, including several architectural refactoring patternsDelves into details of database decomposition, including the impact of breaking referential and transactional integrity, new failure modes, and more Get full access to Monolith to Microservices and 60K+ other titles, with a free 10-day trial of O'Reilly. There are also live events, courses curated by job role, and more. How do you detangle a monolithic system and migrate it to a microservice architecture? How do you do it while maintaining business-as-usual? As a companion to Sam Newman’s extremely popular Building Microservices, this new book details a proven method for transitioning an existing monolithic system to a microservice architecture. With many illustrative examples, insightful migration patterns, and a bevy of practical advice to transition your monolith enterprise into a microservice operation, this practical guide covers multiple scenarios and strategies for a successful migration, from initial planning all the way through application and database decomposition. You’ll learn several tried and tested patterns and techniques that you can use as you migrate your existing architecture. Ideal for organizations looking to transition to microservices, rather than rebuild Helps companies determine whether to migrate, when to migrate, and where to begin Addresses communication, integration, and the migration of legacy systems Discusses multiple migration patterns and where they apply Provides database migration examples, along with synchronization strategies Explores application decomposition, including several architectural refactoring patterns Delves into details of database decomposition, including the impact of breaking referential and transactional integrity, new failure modes, and more At present, there are plans to translate the book into Portuguese, German, and Chinese (simplified), in addition to creating an audio book. I'll update this page with details of when these editions are available, along with information of any other confirmed translations. Topics Should you migrate to microservices, and if you should, how do you prioritise where to start How do you incrementally decompose an application Discusses multiple migration patterns and where they apply Delves into details of database decomposition, including the impact of breaking referential and transactional integrity, new failure modes, and more The growing pains you'll experience as your microservice architecture grows Table Of Contents What Are Microservices? The Monolith On Coupling And Cohesion Just Enough Domain-Driven Design Understanding The Goal Why Might You Choose Microservices? When Might Microservices Be A Bad Idea? Trade-offs Taking People On The Journey Changing Organizations Importance Of Incremental Migration Cost Of Change Domain-Driven Design Reorganizing Teams How Will You Know If It's Working? Pattern: The Shared Database Pattern: Database View Pattern: Database Wrapping Service Pattern: Database-as-a-Service Interface Transfering Ownership Data Synchronization Pattern: Synchronize Data In Application Pattern: Tracer Write Splitting Apart The Database Pattern: Split Table Pattern: Move Foreign-Key Relationship To Code Transactions Sagas More Services, More Pain Breaking Changes Reporting Monitoring and Troubleshooting Local Developer Experience Running Too Many Things End-to-end Testing Global Verses Local Optimization Robustness and Resiliency Orphaned Services Jump to ratings and reviewsHow do you detangle a monolithic system and migrate it to a microservices architecture? How do you do it while maintaining business-as-usual? As a companion to Sam Newman's extremely popular Building Microservices, this new book details a proven method for transitioning an existing monolithic system to a microservice architecture.With many illustrative examples, insightful migration patterns, and a bevy of practical advice to transition your monolith enterprise into a microservice operation, this practical guide covers multiple scenarios and strategies for a successful migration, from initial planning all the way through application and database decomposition. You'll learn several tried and tested patterns and techniques that you can use as you migrate your existing architecture.*Ideal for organizations looking to transition to microservices, rather than rebuild*Helps companies determine whether to migrate, when to migrate, and where to begin*Addresses communication, integration, and the migration of legacy systems*Discusses multiple migration patterns and where they apply*Provides database migration examples, along with synchronization strategies*Explores application decomposition, including several architectural refactoring patterns*Delves into details of database decomposition, including the impact of breaking referential and transactional integrity, new failure modes, and more 536 people are currently readingDisplaying 1 - 30 of 98 reviewsNovember 26, 2019Sam Newman did it again. He has written a very good book on microservices.The one which is technology-agnostic & in the same time - very practical.What I like most is that Sam doesn't try to avoid answering uncomfortable questions - e.g. what to do when we have queries spanning across separate storages. Another positive fact is that we're not getting 100th description of what is CQRS & Eventsourcing.What else? There's no zealotry, no expressed preferences regarding any particular tools (except Sam's love towards FaaS ;>), examples presented are good enough (not really deeply into the details, but on a sufficient level).What did I miss? Some more explicit statements regarding data redundancy (pragmatism over normalization), helpful conventions (append-only approach & immutability of selected data - this was partially covered in the chapter about deletes).Strongly recommended. Short, but a very decent book.P.S. The version I've got was sponsored by nginx (2019.10) - it may differ slightly from the version which will be mass-printed.January 27, 2020Overall I found this book to be an excellent, practical guide to approaching a monolith decomposition, though I have a few issues with it. The Good:- Newman starts by presenting all of the reasons why you might want to do microservices and how you could solve them WITHOUT doing microservices. This was the main complaint with "Building Microservices" which presents microservices without being too critical about when one would want to avoid microservices. - There's a great section describing how one might help their organization to make a change to microservices. This section builds nicely on Newman's second chapter of "Building Microservices", "The Evolutionary Architect". - There are some good decomposition patterns, notably: branch by abstraction and the strangler fig pattern. - There is an EXCELLENT section on the growing pains one might encounter when they start adopting microservices. I found this section to be particularly useful because it's leveraging Newman's considerable experience as a consultant, in which he's seen lots of different companies adopt microservices and encounter problems. - Newman focuses on core, long-lived aspects of migrations instead of on specific technologies (which will be outdated in a year), with just enough technology examples to help one connect concepts to real world examples. The Not So Good:- The actual decomposition recommendations almost entirely ignore the issues brought up in the "Growing Pains" chapter. One huge issue which microservices bring up is that network calls have a different guarantee than databases: they may fail, they aren't transactional, they may timeout, etc. This causes big problems with two proposals in the book: - Dual write migrations - A migration strategy in which one writes to both systems using rest calls, migrates data, and then changes reads to the new source of truth. The big issue with this strategy is keeping both stores consistent since calls to either system can fail or timeout. In my experience, this is one of the hardest parts of migrations, and it's barely handled as written. I would have loved to see more examples of how companies handled this, from Newman's experience. - Orchestrated Sagas - A similar issue arises in Orchestrated Sagas. There may be partial failures on any part of the way with both the inbound saga and the "compensating transaction". The book suggests using choreographed transactions where possible but doesn't bring up this danger of orchestrated transactions. I'm wondering if it would have been better to restructure the book to first present the unique issues which microservice architectures encounter and then present decomposition techniques which lead to architectures which are resilient to these issues. - The decomposition examples given are pretty straightforward. This is probably to make the points very clear, but it would have been great to have at least one example of a decomposition which was tricky or which wasn't worth decomposing at all. Newman mentions that such examples exist, but doesn't do any further examination. Overall: Overall, a very practical guide to migrating from a monolith to microservices, and (importantly) why you might not want to.April 5, 2020One word: BORINGThe book is basically 2~3 chapters and bunch of filler chapters.If you have some experience or got your hands dirty already with microservices, then probably the majority of the book is known to you, the material you read and watch online on daily basis covers the majority of the book.If you don't have any experience with microservices AND you've been living in a cave to avoid online material about microservices then it might be a good book for you.June 12, 2020What I liked a most in the book is that is short and dense. It have everything in proper amount of length, giving a good overview of the problems, and provide some of possible solution to them with a bit of “what I would do”.Really good book. June 26, 2020Practical advice on how to move to Microservices. It also discusses the reasons you would want to migrate your app to a Microservices architectural style and even gives alternatives that might help you achieve the same goal without migrating. I liked the realistic examples, trade-off analysis.I enjoyed Chapter 4, on Decomposing the Database the most, because I think that's probably the hardest part when migrating to Microservices (and the one that gets the least amount of coverage in many resources). The patterns cover problems we've all seen in the wild and the solutions are pragmatic and useful.September 17, 2023Having stable interfaces between services is essential if we want independent deployability.Quite insightful and practical. A big part of the book is patterns that can be applied when transforming from a monolith to microservices. There were some patterns that I skimmed through faster (mainly because I had good familiarity with them already), but it didn't take away from my overall enjoyment.February 1, 2022I confess that the title made me think of a very boring and strictly tactical book. However the book starts with a strategic view of microservices, specifying the benefits and drawbacks of the approach regarding large escale engineering and startups as well. Later on, SN covers some very useful patterns when it comes to design microservices from unstructured monolithic architectures. Furthermore, it covers precisely the challenges of tackling databases when decoupling your monolithic architecture. Finally, I really appreciated the sobered and mature view of SN considering the impacts os scaling your microservices architecture on large scale software engineering taking into account the cognitive load that some organizations might face if they are too much permissive. This is an essential book for architects and software engineers that wish to improve their knowledge on possible challenges and solutions dealing with microservices. In my point of view this book is paramount for technological leaders - CTOs, Chief Architects and so on - that are facing a technological growth and intend to employ some evolutionary software architecture strategy. November 29, 2020A nice book with lots of recepies, patterns and best practices for breaking monoliths into microservices. Not too detailed into tech stuff, but still good anough.May 10, 2020I went through many of mentioned problems in hard way. Anyway, it was a good opportunity to stop and think. I like that this book does not push microservices for everything. Architectonic hints described in the book could be handy for any modular system.I got a free copy via nginxOctober 14, 2022July 16, 2021A great book presenting patterns for transforming a monolithic system into a microservice architecture. The Sam Newman’s exposition is clear (but dense!), very pragmatic and grounded in reality, reflecting real-life experience in countless projects. It gives me a lot of confidence that a good chunk of the initial chapters is spent giving you reasons not to choose microservices: all the pros and cons are evaluated, and alternatives are proposed. This type of architecture is not for everybody, especially not for startups.If after all the warnings you’re still convinced that microservices are the way to go, the author proceeds by defining basic concepts and outlining the general plan for a successful migration. DDD is presented as an invaluable tool for identifying service boundaries, and several other aspects are also explored: cultural changes, team satisfaction, ownership, etc. because technical matters are not the only subject that needs to be considered.The core of the book is in chapters 3 and 4, which present a catalog of high-level patterns for splitting a monolith both from the service’s perspective and from the database’s perspective. The solutions to common problems are explained at different levels of detail: some involve fiddling with networking and infrastructure, others modifying service code via refactoring, and others propose changes in the database or its tables – but never going into actual implementation details.Several of the patterns felt familiar; I’ve probably applied them in some capacity without knowing they existed (but I wish I knew them before!) I was very satisfied with the patterns and the methodology, it encourages incremental, evolutionary changes in architecture while offering multiple alternatives; sometimes the author shows his preference for one over the other but he’s emphatic that all depends on the context of each problem. For an example of the plethora of options available for each situation, take a look at the discussion of how to split reference data in a database.A couple of the patterns present somewhat surprising solutions, I really liked this out-of-the-box thinking, away from dogmatisms and from the tyranny of batch processing and monolithic databases, making a good case for event-based microservice architectures. The final chapter deals with practical considerations and tips when the number of services starts to grow, and when to expect the problems to pop up. Interesting stuff to have in mind. Also in this chapter and in the whole book in general, the author recommends open source tools for different needs, I’ll list the most useful for my own future reference: change data capture systems, service meshes, GitHub Scientist, FlywayDB, SchemaSpy, ELK, Jaeger, pact.io.A final word of advice: bear in mind that this book is not a standalone reference. To gain a good understanding of the details of implementing a microservice architecture, one needs to also read “Building Microservices” by the same author, which I intend to do – the perfect companion for this volume. Needless to say, I highly recommend this book to any architect considering jumping into the microservice bandwagon, but doing so in a responsible way.January 22, 2020A great thing about this book is that it covers a lot of the non-technical prerequisites that need to be considered within an organization, before even bothering to attempt to migrate to a microservices architecture. Basic questions about “why” an organization would even want to do that are not-so common sense, but essential. Without knowing why your organization wants to make such a move most likely condemns the “IT department” to merely creating the legacy systems of tomorrow.There is also coverage of methods to help advocate for such prerequisite organizational change, which is probably not something your average software developer is familiar with, so a very valuable read indeed.The patterns for migrating from monoliths to microservices in Chapter 3 include some that are just useful pattens in any system migration or modification context, but the coverage is clear and reminded me of the times when I’d employed similar techniques myself. As is the case with design patterns, having a common vocabulary for them is a valuable aspect.Chapter four covers decomposition of your classic relational database schemas into more microservices friendly granularity, without glossing over the trade-offs to be understood.The last chapter overviews some types of issues that one might experience during an incremental journey to more and more microservices and perhaps a growing organization (if you are lucky to be part of a successful one!)I read the Building Microservices book first, but found this one useful for its focus on migrating from a monolith / shared database nightmare to microservices. I’ll certainly take care to pull this back up when I’m facing some kind of migration issue, for inspiration or ideas I might forget in the meantime. That said, if you can properly modularize your system(s) then you might be better to not even think about microservices, a point made in both this book and the earlier one.I read the 1st edition - there’s several editing / typo problems throughout (mostly chapters 3 and 4 I think), but not so bad the content can’t be understood.March 2, 2021Um apanhado de boas práticas e exemplos adquiridos pelo autor. A parte técnica é bem superficial, mas apresenta boas referênciasMarch 29, 2021This book has some good advice on how to manage an application or database migration in small steps. The first and last chapters are just an introduction to microservices and their problems. Nothing new especially for those who have already read "Building Microservices".The second chapter helps us to plan the migration including the soft skills required. But more interesting than that, it shows us that depend on the problems we are trying to solve there are some easier and cheaper alternatives to microservices that do necessarily involve code. For example, changing the organizational structure or the infrastructure.The third and forth chapters talk about the patterns. This should be the most important part of the book but I didn't like how the patterns were organized. Comparing to other "patterns" books like "Design Patterns" and "Enterprise Integration Patterns", this book is much more disorganized. One of the most valuable thing about the patterns is that when you read about one of them, you can also have a good overview about other related ones. Here, the patterns are sometimes compared only in one way. For example, if patterns A and B are related and B is presented last, the chapter that talks about B will compare it to A but not necessary the other way around. That way, you can only see the relationship among them if you read them in order.Besides that, some patterns seems not to be very useful. The most important ones are listed and explained in this talk was made just before this book release: think it is something between 3 and 4 stars. I will rate as 4 because it is not easy to find material about it.architecture enterprise-architectureMarch 15, 2024"Monolith to Microservices" is a highly insightful book that diligently tackles the complex process of transitioning an application from monolithic architecture to microservices. It bridges the gap between theory and practice, bringing valuable pragmatic recommendations to the table.The book excels in delineating modern architectural patterns, providing an updated framework for individuals grappling with contemporary software development needs. Not only does it discuss the 'how-to' extensively, but it also lays emphasis on the 'why', offering readers the understanding to make informed decisions rather than blindly following best practices.A key strength of this book lies in its balanced view. The author doesn't merely paint an idyllic picture of microservices but delves into potential pitfalls and challenges too. This clear-eyed perspective extends beyond pure technical implications, touching on organizational, cultural, and operational pitfalls that one might encounter during the transition.This feature turns the book into more than just a technical guide; it becomes a holistic tool for organizations seeking to navigate the rough waters of infrastructure transformation. The author's articulate style blended with relatable examples and practical tips aids in transforming esoteric concepts into comprehensible knowledge.In conclusion, "Monolith to Microservices" is a must-read for anyone involved in or looking to embark on the journey of transitioning from a monolithic architecture to microservices. It is woven with wisdom, cautionary tales, and helpful strategies that will guide you effectively through the complexity of such a transformation.March 14, 2023This book is gold. I wish I had read it when I was struggling to break down a legacy monolith into microservices. A lot of the problems I faced were actually covered in this book, and even if I couldn't have avoided them, I would have been better prepared for them. The book isn't just about microservices, but it also covers how to break down your codebase and persistence layer to make them more modular. It even helps you to consider whether you need to do this in the first place and the cost of doing it.I think it would be great if this book had a second edition with more practical examples. For instance, most people are aware that to decouple and migrate part of your database to a new microservice, you need to copy the data and start using it. However, it's actually more challenging to do than we realize, especially if you want to achieve zero downtime. Simple backup and restore, along with some synchronization strategy, may not be enough. It can be a real challenge to avoid a maintenance window while ensuring that the data is consistent between both databases (consider using write-ahead log for synchronization and MD5 checksum hash function for testing).I'm currently reading Software Architecture: The Hard Parts, and I think these two books complement each other pretty well.March 3, 2021It's so hard to write a good book about software architecture. I think you have two options: digging deeply into the technical details or trying to be technologicaly agnostic and relying only on "abstractions". This book is of the latter type - word "Docker" is used only once and word "Kubernetess" is used less than five times :) And this is a really good book.What I liked:- Simplicity. All the patterns short and well explained. Moreover, we always have a context of usage - it's easier to put it in the "real-life" scenario.- Pragmatism. I like the idea of slider instead of button for description of making changes. Nothing is for free and we should always look at architecture migration in this way.What I didn't like:- The shallowness of some topics. For example, the phrase "extract your services based on bounded contexts" is in self a topic for multiple books. However, I understand that exploring such problems in this book would increase the number of pages dramatically :) I accept it.April 10, 2023El libro es altamente práctico y está estructurado en seis partes, cada una enfocada en un aspecto diferente de la migración a microservicios. En la primera parte, Newman presenta una introducción general sobre las arquitecturas monolíticas y de microservicios. En las siguientes partes, se enfoca en aspectos específicos, incluyendo cómo dividir una aplicación monolítica en microservicios, cómo implementar una arquitectura de microservicios, cómo diseñar y mantener bases de datos distribuidas y cómo manejar problemas como la monitorización , la seguridad y los errores.A lo largo del libro, Newman incorpora numerosos ejemplos de la vida real de su experiencia trabajando en diversos proyectos y organizaciones. Las historias sugieren que aunque la migración de una arquitectura monolítica a una basada en microservicios es un proceso difícil, el resultado final puede ser positivo al mejorar la escalabilidad, la flexibilidad y la agilidad de las organizaciones.May 6, 2025Most useful concepts were:- Purpose of microservices being: independent deployability; modeling of business domain or functionality, rather than a technology; having ownership of data.- Information hiding: keep boundaries stable, and only change the internals.- Try to salvage the monolith before switching to microservices. Split the monolith into modules first.- Strangler Fig pattern for migration.- Branch by Abstraction pattern for migration.- Using parallel runs (n-version programming) for high-risk migrations.- Coping patterns to avoid splitting the existing database into logically separated schemas (e.g. views; database wrapper service; database-as-a-service).- Differences between the monolith exposing a data API endpoint to the microservice vs. the other way round.- Tracer write: start syncing small pieces of data with a phased switchover. Until complete switchover, the databases will have syncing delay but with the goal of eventual consistency.- Long-lived transactions and sagas.October 28, 2020Another great microservices book from Sam Newman. I felt this is a sequel to his earlier book, Building Microservices. Sam updated some concepts from recent microservices evolution such as choreographed vs orchestrated Sagas, added infrastructure evolution such as Kubernetes. I loved the depth he covered the monolith database refactoring and the simple but elegant solutions he proposed using schemas and views in existing database engine for refactoring. I'm a fan of Sam's pragmatic approach when introducing new concepts that works very well for startups and big corporations. He also delved in to details about the organizational aspects of microservices and the distributed systems concerns that microservices might bring. If you liked reading his earlier book Building Microservices, then this is a highly recommended sequel helping in implementing that.April 7, 2021This book seems to be a step 1 if an organization is flirting with the idea of moving from a monolith to microservices. I like the fact the author is honest about evaluating your needs before allowing yourself to be swept by the herd. Since I am a part of an organization which has already made the decision to switch to using microservices (and rightly so!!), it's interesting to know that some issues are quite widespread regardless of which organization you work for and actually see them in front of myself. The author has identified those issues and has given solutions which are actually being implemented by my team (eg: Strangler Fig, Shims). But at some point, one might realize that the author has written things that are quite obvious and seemed repetitive (parity with legacy functionality, end-to-end integration testing, breaking changes). It kind of became a drag in the 5th chapter.March 17, 2023Really great book.It does not concentrate on any specific technological stack, but gives very good advice on what to be aware of when moving to microservices, what problems they could solve, and decide whether such migration is the right step for you.Sam is not "preachy" about microservices, and honestly states possible problems, and cases where migration won't help you.While it's impossible to give advice that will work for 100% of people, author does provide a wide selection of options for migrations, with considerations for each one, so that one can compare their situation with "hypothetical" one, and adapt decision for their case.If you have a monolith currently, or thinking about migration, or you're already part of microservice company, but want to know more about challenges of microservice architecture - this will be a great read.August 18, 2020Pure excellence - must read It is an amazing book. It covers everything from start to end. If you want to travel the road of Microservices. It’s structured in a way that fits the mental model of monolith developers but to evaluate Microservices architecture. It’s going to serve a glossary of what to do & what not to do when doing Microservices. Starts with simple 3 questions, during the journey you would feel yeah it feels very real. But without single line of code. You will get answers along the way, from people problems, team ownership, tackling database, importance of DDD and suggestions on tool one has to learn along the way. Thanks Sam Newman for writing such an amazing book. September 8, 2020A very well written book, which shows in every page the experience of the author on the topic. It really puts into perspective the huge task of moving to microservices from a monolith, the most common pitfalls, and provides possible solutions to them. The book is mostly useful as a reference for your journey of moving to microservices, it is not an absolute proven guide of not failing, but more a collection of very well organized experiences that will help you avoid most common problems and kind of guide your decision making. It is good that the author emphasizes at the start that microservices is not for every situation, and spends some time explaining how you can figure this out for your current situation before you start.August 25, 2021I think this book gives a great overview on how to start a successful Monolith to Microservices transition. The part that I liked the most is that the Technical Project Management techniques described in this book can be applied to any large technical undertaking.Another thing that I really appreciated is that book talks about how Microservices can't be a silver bullet and end coal by themselves and how to identify what problem they can solve if any.However, to actually lead the process of splitting the monolith before reading this book I felt the reader needs to have a good understanding of Domain-Driven Design. The book covers it very briefly.August 26, 2021This book was illuminating to me in terms of laying out a process for getting from a to z. I'm currently in the middle of such a transition in my work as a software engineer and I learned a lot about the why and the how of to do so. I enjoyed the lack of real code examples instead the book focuses on describing concepts through simple diagrams. The work of breaking apart monoliths is not obvious and there are a lot of pitfalls one can find themselves in if you don't have the experiences. I now have a lot more clarity on what the right approaches are in a wide variety of situations. The book is also quite short and readable which made it easy to digest.November 6, 2021Definitely a book every software engineer must read! Newman shows how complex is the job of restructuring a monolithic application in a microservice one, where you can fail and how much. Correctly the author put the doubt into your mind if it is the correct choice for you to follow the microservice approach or not, or if you are ready for that. There are so many challenges that the team and his organisation must be well aligned with that challenge! Never do the transformation all together, but design and follow an iterative and evolving path. Complex software will become even more complex with the transformation, and only a strong discipline and good organization will make you successful!January 23, 2022Book covers various patterns that teams can use in transforming monoliths to microservices. It lists out and has high level details on what could be possible path of migrations in different conditions. Sam has also given his personal preferences after every pattern. For a team that has started to ask whether to migrate to Microservices and if yes, then what could be general guidance, this is a good work. It won't take long to read it end to end. Key takeaway would be - list of patterns and pros and cons of each one of those. You can then start your in-depth analysis of your systems by keeping all of this information at the forefront. It can make your decision making exercise more objective.November 9, 2020Why Sam Newman wrote this book? (IMO)Because as Martin Fowler wrote, "You must be THIS tall to use microservices".When should I move to microservice architecture?What are the pros and cons of microservice architecture?Which module of my system should I move as first?What is the process of refactoring to microservices?What are the best practices for refactoring to microservices? What new problems will occur after moving to microservice architecture?This book will help you grow to "THIS" tall by answering those and other questions.March 24, 2021Honestly in the past years I have read a lot of titles about microservices architecture but this one turned out to be actually useful for me and my job. For the last two years I was desperately trying to get rid of monolithic structured applications and this book gave me the best approaches to do that! It is fairly simple but really practical and useful! Congrats to Newman! Although there were many thing I already knew, there were also some observations and simple advices that can definitely change your workflow whenever you are facing such issues.Displaying 1 - 30 of 98 reviewsGet help and learn more about the design. Download Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith PDF How do you detangle a monolithic system and migrate it to a microservice architecture? How do you do it while maintaining business-as-usual? As a companion to Sam Newman’s extremely popular Building Microservices, this new book details a proven method for transitioning an existing monolithic system to a microservice architecture. With many illustrative examples, insightful migration patterns, and a bevy of practical advice to transition your monolith enterprise into a microservice operation, this practical guide covers multiple scenarios and strategies for a successful migration, from initial planning all the way through application and database decomposition. You’ll learn several tried and tested patterns and techniques that you can use as you migrate your existing architecture. • Ideal for organizations looking to transition to microservices, rather than rebuild • Helps companies determine whether to migrate, when to migrate, and where to begin • Addresses communication, integration, and the migration of legacy systems • Discusses multiple migration patterns and where they apply • Provides database migration examples, along with synchronization strategies • Explores application decomposition, including several architectural refactoring patterns • Delves into details of database decomposition, including the impact of breaking referential and transactional integrity, new failure modes, and more...