O'REILLY®

Free Chapter

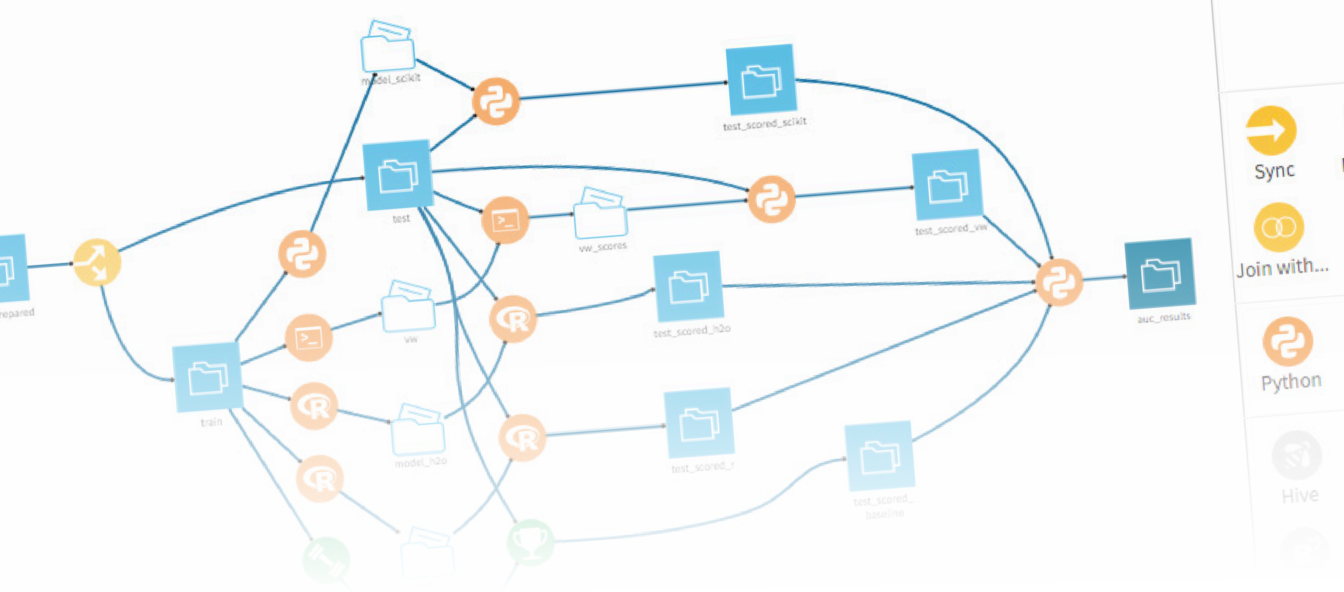
compliments of

data iku

Foundations for Architecting Data Solutions

MANAGING SUCCESSFUL DATA PROJECTS

Ted Malaska & Jonathan Seidman



YOUR [SCALABLE. AUDITABLE. SECURE] PATH TO ENTERPRISE AI

Dataiku is the platform democratizing access to data by providing one simple UI for data connection, wrangling, mining, visualization, machine learning, and deployment.

For IT teams, Dataiku brings:

- A consistent software layer for analysts and data scientists, allowing for flexibility and elasticity in underlying data architecture.
- A scalable, controlled way to push models into production.
- Centralized insight into model life cycle.
- Enterprise-grade governance, including auditability and permissions.

GET THE COMPLETE DATAIKU DATA SHEET

Foundations for Architecting Data Solutions

Managing Successful Data Projects

This Excerpt contains Chapter 3 of the book *Foundations for Architecting Data Solutions*. The complete book is available at learning.oreilly.com and through other retailers.

Ted Malaska and Jonathan Seidman

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Foundations for Architecting Data Solutions
by Ted Malaska and Jonathan Seidman

Copyright © 2018 Ted Malaska and Jonathan Seidman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nicole Tache and Michele Cronin
Production Editor: Nicholas Adams
Copyeditor: Octal Publishing, Inc.
Proofreader: Sharon Wilkey
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2018: First Edition

Revision History for the First Edition
2018-08-29: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492038740* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Foundations for Architecting Data Solutions*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Dataiku. See our statement of editorial independence.

978-1-492-03874-0

[LSI]

Table of Contents

Managing Risk in Data Projects. 1
Categories of Risk 1
Technology Risk 1
Team Risk 2
Requirements Risk 2
Managing Risk 2
Categorizing Risk in Your Architecture 2
Technology Risk 5
Strength of the Team 6
Other Teams 7
Requirements Risk 8
Tying This All Together 9
Using Prototypes and Proofs of Concept 12
Build Two to Three Ways 12
Build PoCs and Then Throw Them Away 12
Deployment Considerations 13
Using Interfaces 13
Start Building Early 15
Test Often and Keep Records 15
Monitoring and Alerting 16
Communicating Risk 17
Collaborate and Gain Buy-In 17
Share the Risk 18
Using Risk as a Negotiation Tool 18
Summary 19

Managing Risk in Data Projects

Humans excel at worrying about things, but normally we worry about the wrong things. As a child, Ted wasted what should have been many hours of sleep on planning an escape route when zombies attacked. People fear a lot of things that, whether grounded in reality or not, will most likely never affect them. A perfect example is that people are more afraid of sharks or terrorist attacks, when the chances of dying from heart disease or a car accident are significantly higher.

This is also true when it comes to software development. Identifying what to worry about and what to not worry about is an extremely powerful practice when implementing successful projects, especially when you're working with new technologies. If managed properly, risk can be an opportunity—if everything was known at the start of a project, what would be the fun in that?

In this chapter, we discuss how to manage risk after completing the software selection process and moving on to implementing a project. The focus of this chapter is on helping you set up methodologies and an environment for success by using development principles and strategies for managing and mitigating risk, setting realistic expectations, and providing a guide to building successful teams.

Categories of Risk

Before we delve into the details of risk management, let's first talk about the broad categories of risk that we want to address in project planning.

Technology Risk

Any software project involves risk, and building large, complex distributed data solutions can increase these risks. This is especially true when those projects are based on new and unfamiliar systems. Risk can come from individual components used in implementing the architecture, interactions between components, and so on.

Risk can also come from unfamiliarity with a technology used in designing the system. Fortunately, there are strategies that we can employ to manage and mitigate these risks as we move from design to implementation of our application.

Team Risk

Team risk refers to the risk associated with the team members implementing the data architecture, as well as external teams. This risk can come from the knowledge level and strength of your team, dependencies on external teams, and even potentially disruptive team members.

Requirements Risk

A common source of requirements risk is poorly defined requirements or sometimes poorly defined problems. Another source of requirements risk, particularly with new technologies, are requirements that your team hasn't worked with before. For example, latency or throughput requirements that are higher than anything the team has built before. Scope creep can also be a common source of requirements risk.

In the next section, we go into more details on these risks types and discuss how to manage these risks.

Managing Risk

In this section, we begin with a methodology that uses a high-level model to assign risk levels to components in the system—this has worked well in practice on various projects. These risk levels encompass the different types of risk we discussed in the previous section and provide a framework to quantify and address these risks.

Categorizing Risk in Your Architecture

This approach starts by breaking the architecture into pieces. Figure 1 shows an example of breaking apart a common data storage and processing system.

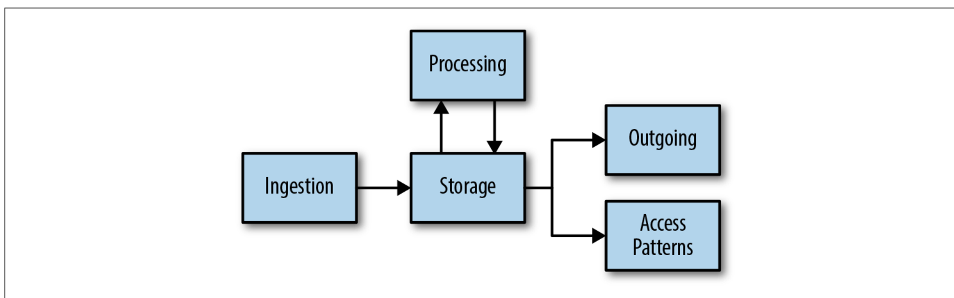


Figure 1. Breaking an architecture into high-level components

In Figure 1, we have an example of breaking up a system into data ingest, data serving, data processing, access patterns, and storage. We can drill down even further to subcomponents within these components, as shown in Figure 2.

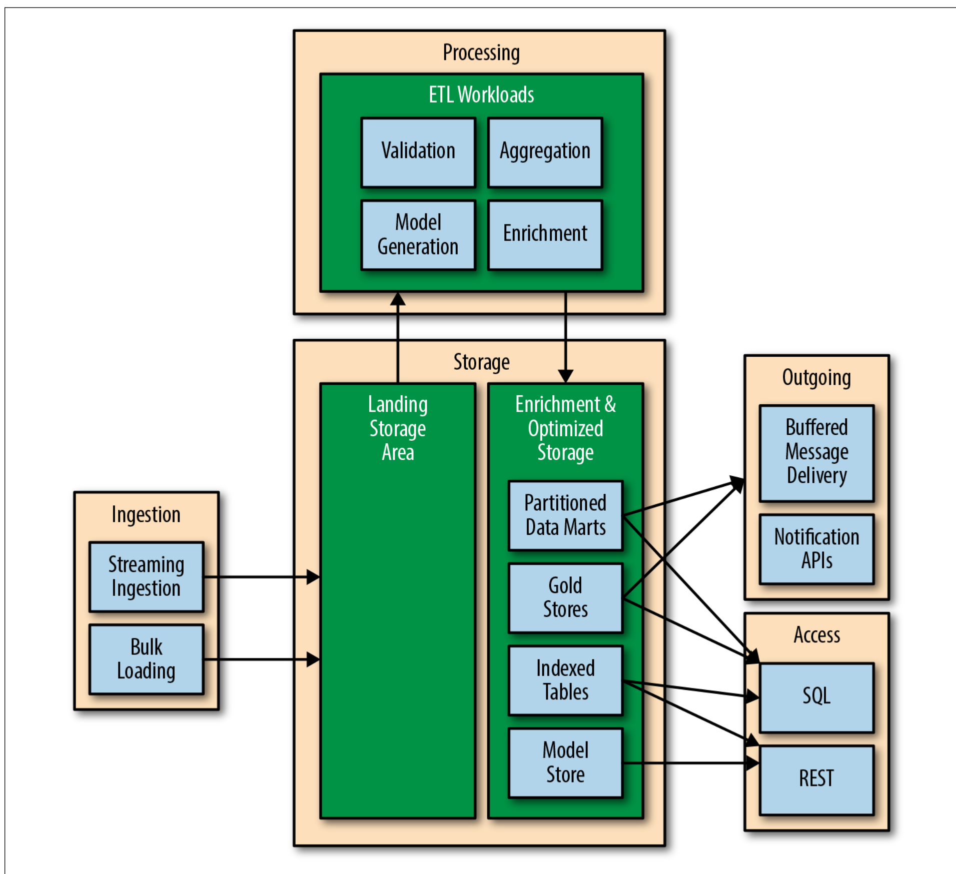


Figure 2. Breaking an architecture into subcomponents

At this stage, we likely won't want to go any deeper in breaking down the system. After we've reached this level, we want to define interfaces to each subcomponent. This will help mitigate the risk of failure in one subsystem affecting the rest of the system. (We examine the subject of interface design later in this chapter.) For now, just think about this as a system of self-contained components that you can develop independently from the others, and that allows the risk of each component to be contained within that component.

The next step is to begin assigning which technologies you're going to use to implement components, as illustrated in Figure 3. You can then assign team members to

work on each component and then apply risk weights to the technologies and to development teams; we discuss this process shortly, starting with technology risk.

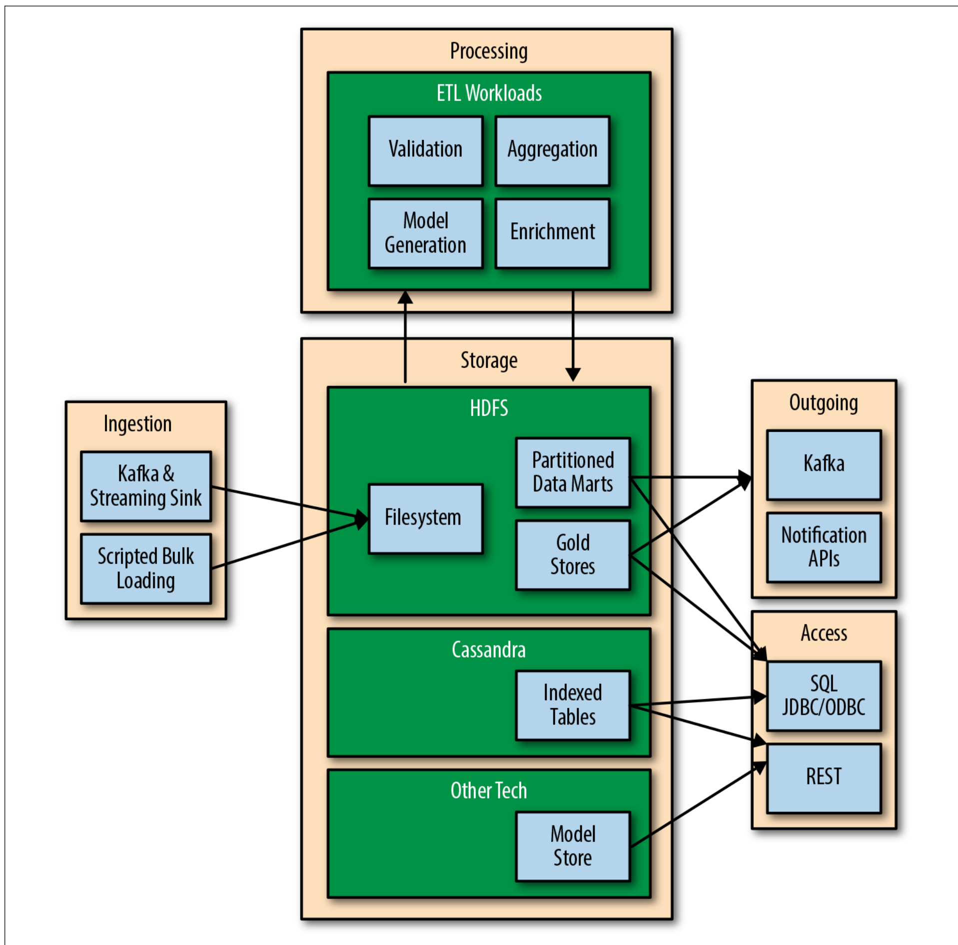


Figure 3. Assigning technologies to subcomponents

Figure 3 shows that we've made technology selections, including the following:

Kafka
: For buffered streaming of data.

Hadoop Distributed File System (HDFS) and object storage
: For initial landing zones and long-term storage for data.

Cassandra
: For high-performance indexed storage.

Other
: This is a placeholder for things like in-memory databases and search-based systems such as Elasticsearch.

For now, you can view these technology selections as placeholders; we dig into detailed reasons to select one technology over another in later chapters. For example, we talk in more detail about storage layer options in Chapter 5, and data processing options in Chapter 8. The main idea here is to have technology selection defined within the initial design, with the knowledge that if the technology is determined to be unsuitable, we can just swap it out. Let's go through an example.

Suppose that you select a NoSQL like Cassandra or HBase to be the base store for a time-series database. Everything works fine initially, but you run into issues as you move along in the life cycle of the project; for example:

- Cassandra can't store enough data on any given node, and the price becomes an issue.
- HBase or Cassandra background processing (e.g., compaction) of data starts to affect performance.
- Query time aggregation is turning out to be too expensive as the number of unique metrics reaches a certain point.
- Performance is fine, but pricing with a vendor becomes a concern.

So, there could be a number of reasons why your initial technology selection doesn't work out, but fortunately if you follow our recommendations in Chapter 4 on using interfaces in your architecture, you can make these changes without causing significant disruption to your design.

Technology Risk

Risk weighting for a team or technology is not an exact science, but normally gut feeling can go a long way. For example, if you have used a given technology in production before and your team has considerable experience, that technology should have a lower risk rating. Conversely, you should, of course, assign a technology with which you have little or no experience a higher risk rating.

Also, knowing a technology involves many levels of knowledge. As an example, consider SQL. Your team can have years of experience with SQL but might not have a lot of experience with SQL on newer "big data" query engines like Hive, Spark SQL, Cassandra CQL, or Impala. Not all query engines behave the same way, and different engines might or might not support specific SQL features. Understanding these capabilities and limitations is key to using one of these systems and extending beyond just

knowing the query language. When we say that we know a technology, we need to know it from top to bottom.

> ### Sometimes, the Best Tool Is the One You Already Know
>
> Keep in mind that there are almost always multiple tools to solve a particular problem. As Chapter 1 discusses, often one of these tools might be one that you're already using. If multiple organizations, including your own, are having success with a tool, it seems like a safe bet that the tool is a solid choice for your architecture.
>
> The important question is how well your team understands the tool, as well as the requirements of your project. This will help in understanding if the tool is a good choice for your application. A hammer is great for driving nails into a wall. It can also serve as a can opener but is definitely not optimal for that task, and it's not so good at washing windows.
>
> In short, knowing your tools and what they are good for is key to being successful with those tools.

Strength of the Team

Next, we should look at the risk level of a team. This will come down to your personal knowledge of the abilities of team members as well as the team's history of completing tasks and meeting deadlines. If you're working with new resources or less-experienced team members, that will likely mean a higher risk weighting for that team.

Of course, regardless of experience levels, each team has different types of people with different skill sets. The following describes some of the types typical of development teams, and a well-rounded team should have at least some of the following personalities:

The cleaner
: This person has a meticulous attention to detail and will ensure that the project has complete test coverage, code is fully version controlled, and so on.

The prototyper
: This person is not afraid to experiment and investigate new software. Their role is to test out approaches and hit risk areas before it is too late to alter the design or approach.

The workhorse
: These folks serve a critical role because they are the ones who will get most of the work done.

The highly flexible
: These are the people who are always eager to learn and grow and are able to adapt quickly. These folks can facilitate bringing the rest of the team along as the project progresses.

The negotiator
: Dealing with project management or external groups is an important task. Not all developers enjoy this style of communication. It takes a special resource to be able to both understand technology and understand how to work across teams to deliver projects.

Additionally, certain personality types are potential red flags for your team and can have direct impacts on your project risk:

The "cowboy" coder
: Anyone who's worked in technology for any period of time is familiar with this type—these are the developers who prefer to go their own way and work on their own. This type is often a very efficient and talented programmer but also not generally a good team player. They can typically get more work done than your average programmer, but often this work won't follow coding standards or be well documented. Sometimes the cowboy can be an asset, but often their inability to work with a team and conform to standards or processes becomes more of a liability.

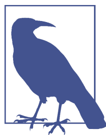
The toxic personality
: Every so often a team will have a member who, sometimes inadvertently, is disruptive to the team dynamics. This can be caused by a number of factors—sometimes these people are argumentative or are convinced they know the best way to do everything. The paradox is that these are often talented and productive individuals, but their impact on team morale and productivity outweighs these talents. Avoid these types—even less talented folks are better as long as they fit well within your team. If avoidance isn't an option, it's important that you ensure that you learn how to manage them.

In addition, it's good to have enough team members so that there are at least two people on each component. In addition to ensuring better coverage for components, having team members be accountable to someone else can often produce better outcomes.

Other Teams

In addition to the risk associated with the project team, there's also the risk of interacting with one or more external groups. This is particularly true if those groups are working on components or systems that are critical to the work of your team or, conversely, where the work of your team is important to the success of external teams.

Whenever your success is directly tied to another group, be mindful and respectful. Having well-defined responsibilities and requirements is important to help achieve this. Additionally, well-defined and documented software interfaces can be key when designing software that needs to interact with systems developed by other teams.



We talk more about interfaces later in this chapter as well as in Chapter 5.

Requirements Risk

After team risk and technology risk, we have the risk level of requirements. This risk can come in different ways. One way is if requirements are vaguely defined, which might mean that later adjustments are needed that can introduce new risks. Another way is if there are requirements that your team hasn't worked with before; for example, service-level agreements or latency goals.

Having good functional requirements can help address these risks. If these functional requirements are treated as a contract that are not technology specific, they can be very effective in keeping the project on the correct path. We discuss some additional approaches to reduce requirements risk later in this chapter.

Breaking requirements into more manageable chunks of work is another way to address requirements risk. As an example of what we mean here, at a client site, one of the authors was presented with a hundred-page requirements document for a processing component. The source of this document turned out to be a previous consulting group that had been brought in to help implement a solution. As it turned out, that consulting group had failed. The approach of "boiling the ocean" taken by this requirements document created a situation in which the client was overwhelmed and unsure where to start; even though the document captured the requirements, it did not facilitate a practical development timeline or help in prioritizing requirements and task. Needless to say, that document was discarded.

After discarding the existing requirements document, the process that was followed was to build high-level requirements that could fit into a single page, along with a big-picture diagram of the intended solution. Figure 3 presents an example of such a diagram. From there, a component was identified that was of high importance, because, if implemented, it would increase confidence that the future solution would work. With this limited scope, it was possible to have a working Proof of Concept (PoC) for the selected component. Based on insight from this PoC, adjustments were made to the high-level requirements and then another component was identified for a PoC. This process repeated itself for about a month, after which a rough PoC for the entire system had been built.

The result is the risks had been identified; the development team understood the requirements at a deeper level; the customer had faith in the design; and with the knowledge learned from the PoC, realistic timelines could be estimated for deployment of a production version of the system.

Tying This All Together

To understand how this all fits together with the system model we've created, let's first recap the areas of risk we've discussed:

Technology knowledge
: What level of experience does your team have with the software being used to implement your application?

Team risk
: How experienced is your team, what personality types does your team have, dependencies on other teams, and so on?

Requirements risk
: What level of risk do your requirements have?

Assigning risk weightings

After performing an analysis of these different areas of risk, we can use this analysis to assign risk weightings to the system model we created. When we have accounted for all of the different risk weighting, we should see something that looks like Figure 4. Note that the figure is grayscale in the print edition, but you're encouraged to use color in your diagrams to better highlight the risk levels.

As you can see by the key, darker boxes indicate a higher level of risk, and lighter boxes represent a lower risk.

You should accompany this by a detailed explanation of why components are assigned a higher risk level. List each component, followed by the risk that the component and associated requirements present. If you give each risk a short title, it can quickly be referenced in discussion.

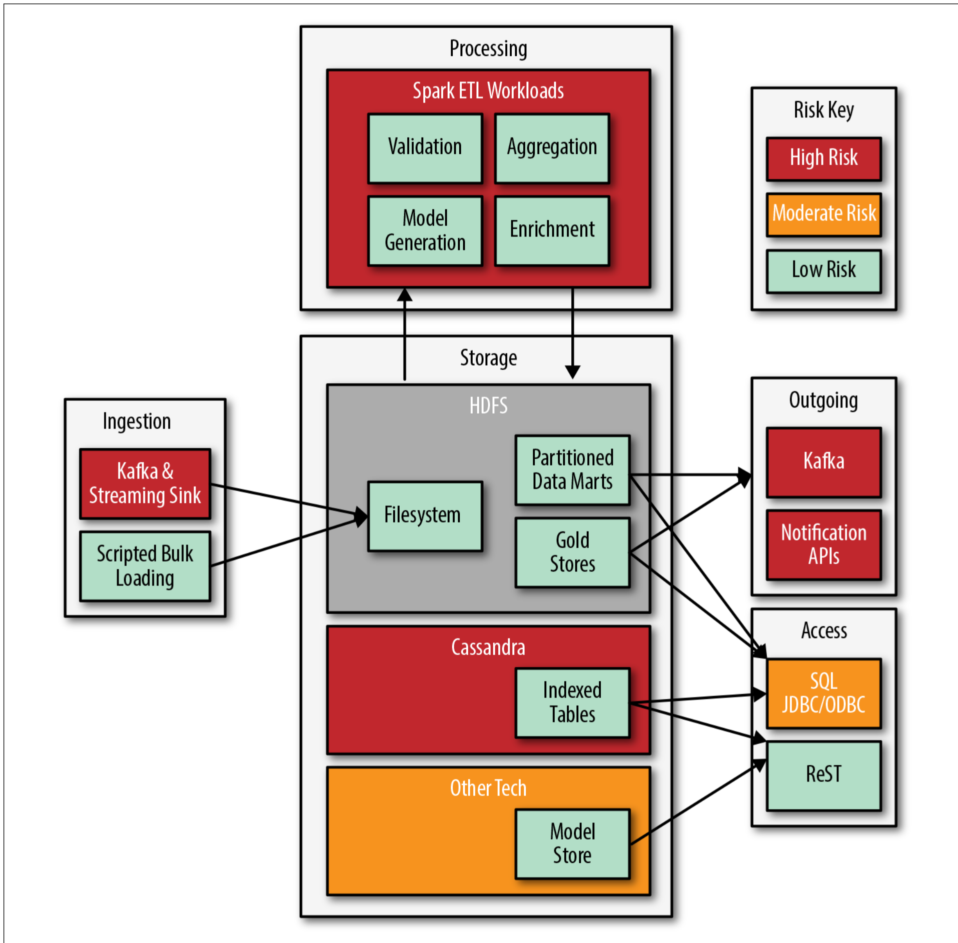


Figure 4. Assigning risk weightings

Following are some examples, with each item in the list notated with the risk type we just discussed.

Cassandra

1. *Technology Experience* (technology risk): We have a limited level of experience with Cassandra within the company.
2. *Data Model* (requirements risk): We need to validate that the data model is correct for our use case.
3. *Uptime* (technology risk): We have SLA requirements that data stored in Cassandra has to be available to users with an uptime of 99.99%, which is beyond our normal SLA experience.

Bulk fetches via Spark

1. *Failure Scenario* (requirements risk): We are unsure about requirements of how to proceed when extracting data fails because of issues on the storage layer.

Kafka Streaming

1. *Technology Experience* (technology risk): This is our team's first time building streaming applications.
2. *Zero Data Loss* (technology risk): There is a requirement for zero data loss, which introduces complicated technical considerations.

Spark ETL

1. *Resource Availability* (team risk): Team members with Spark experience are already committed, which might mean assigning work to less-experienced team members.
2. *Data Model* (requirements risk): Uncertainty about the data model definition might mean having to reimplement code.

Minimizing risk

After we've estimated risk levels, we can take steps to minimize risk. Some of these steps we've already discussed, In the list that follows, we look at some of some of them in more detail:

- Lock down requirements better. As we discussed a moment ago, creating more detailed and precise functional requirements can help minimize requirements risk.
- Make sure that you share your requirements and get buy-in from all stakeholders.
- Create a clear definition of project scope and ensure stakeholder agreement.
- Add additional interface requirements and protocols with external groups. We discuss this in greater detail later, but defining strong abstractions can help reduce architectural risk by making it easier to reimplement components or replace technologies if the need arises.
- Prioritize riskier work. Tackling riskier items earlier provides additional time in case you run into problems.
- In addition to the preceding recommendation, assign riskier technologies to stronger and more experienced team members.
- Use external resources to help address knowledge gaps. You'll learn more about this later in the chapter.

- Use prototypes and PoCs to reduce architectural risk.
- Replace riskier components with less-risky ones. For example, in our hypothetical risk breakdown, we might decide to replace Spark for Extract, Transform, and Load (ETL) processing with a tool that available team members are more familiar with such as MapReduce or Hive.

Using Prototypes and Proofs of Concept

It's important to have backup plans if a component of your architecture fails to meet expectations, but this failure can affect the perceptions of your project or your schedule. What if you could fail faster before a failure becomes too impactful to projects? Let's look at some ways to do just that.

Build Two to Three Ways

The trick is simple: start with straightforward requirements that are wide in their impact and then brainstorm at least two to three ways to meet those requirements. Implement each of your solutions and then run benchmarks. This can help you to select the best of your solutions or come to a realization that you need to explore further. The key here is to build fast—think of these solutions as prototypes. You should learn several things through this process:

- The quality of the documentation of the system you're using
- Performance of the solution: throughput, latency, and so on
- Complexity of the different approaches
- The ability of the team to pick up the technology and develop their opinions of the technology

Build PoCs and Then Throw Them Away

Creating a PoC can be a valuable way to validate a technology or approach. A good way to approach a PoC is to view it as throwaway work and build it as quickly as possible. The goal is to push the requirements and technology as hard as possible within a limited window of time. After a PoC is successful, you should rewrite the code and maybe even have different developers do most of the new implementation. This will provide a couple of things:

More eyes
: Again, having a different perspective can be good validation of your PoC.

Better systems
: The experience and knowledge gained from building the PoC can often help in building better systems.

A common problem with PoCs that you want to avoid is management seeing it and saying "Hey, it works! Ship it." Having said this, it's useful to design PoCs that implement a minimally viable product (MVP) for production deployment. This will allow you to better evaluate the suitability of your solution.

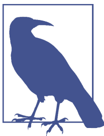
Deployment Considerations

It wasn't too long ago that code would be deployed to servers that were manually configured, requiring long checklists of steps to upgrade or set up a new system deployment. This created considerable risk when it was necessary to deploy new code, update software, and so on because of the lack of tested, repeatable processes to manage changes. Mature organizations now use automation software, including build systems like Jenkins, configuration management systems like Puppet or Chef, and containers such as Docker to manage and automate changes to systems. By using these systems, you can considerably reduce the risk of software deployments.

Although an in-depth discussion of these systems is beyond the scope of this book, the takeaway here is that having tested and repeatable build and deployment systems and processes is key to moving fast and reducing risk. Many production issues are a result of upgrades, configuration changes, library updates, and so forth gone wrong. The more you automate your build and deployment, the more you've reduced your risk of issues occurring during deployments to production.

Using Interfaces

Interfaces are a common and important concept in software development. They provide an important tool to reduce risk in a software architecture by reducing coupling and dependencies between components. Interface design at a project or architecture level is the idea of having different parts of the application agree upon a mode of communicating to other areas of the architecture. A common implementation of this concept is a service layer implementation to serve as the interface between a frontend web application and a backend data store. You could implement this in any number of ways: a Representational State Transfer (REST) interface, a Java interface, a Scala trait, and so on.



The topic of interfaces is important enough that we devote Chapter 4 to discussing the design and implementation of flexible and maintainable interfaces. However, we want to briefly touch here on how interfaces can help to reduce risk in your projects.

So how do interfaces help to address risk? One way is by allowing teams to work independently on different parts of your system. For example, the frontend team can build a dummy implementation of the interface so that they can continue development, testing, and deploying their web application without needing to worry about the progress of the backend teams. Figure 5 illustrates this process.

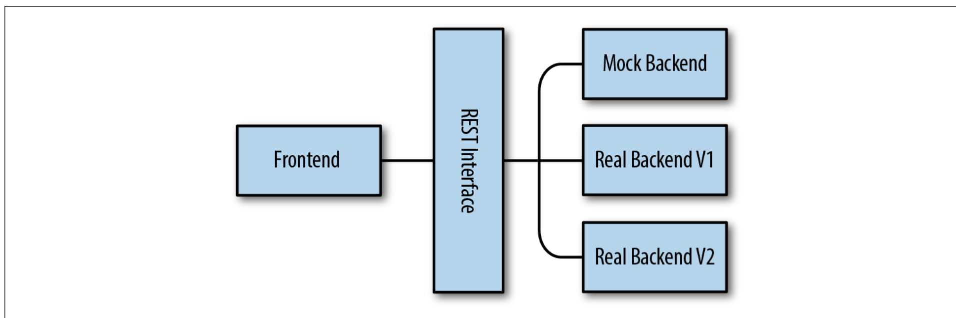


Figure 5. Using mock implementations with interfaces

This also means that the backend team will have the freedom to change up different strategies for storage or execution engines in their implementations without disrupting the work of the frontend team.

Using well-designed interfaces will give both the frontend and the backend teams freedom to change their designs and approaches without having to consult the other team, as long as they hold to the agreed-upon interface that they both share. You can carry out all development, load testing, unit testing, and so on without any dependencies on the other. An additional benefit of this approach can be a significant shortening of development timelines.

A good interface design can also help in communicating risk because it breaks your system into many parts, such as the following:

- Frontends
- Data ingestion and extraction
- Streaming
- Storage
- Databases
- Batch processing
- Real-time processing
- Monitoring
- Auditing

- Disaster recovery

The inherent risk of any one component will have little or no impact on the other parts. This helps in the management of risk by isolating risk to components rather than the entire project. This can provide a couple of advantages for the case in which a component is running into trouble. First, the troubled component doesn't affect the schedule of the other components. Second, when a troubled part is isolated, it provides freedom to change or even totally rewrite it as long as the interface design remains the same.

Start Building Early

Another pitfall that companies fall into is wanting to delay development until after all of the requirements are known, generally because of fears that code will be otherwise thrown away and money and time will be lost. In fact, positive things can come out of starting development as early in the process as possible:

- It will engage your development team, which will lead it to a better understanding of the requirement, which will then help the development team play a more active role in the process of requirements gathering.
- It will help flush out issues with the requirements earlier.
- It can lead to working demos. This is hugely important for two reasons: it can lower the risk level of the project by showing the outside world that real progress is being made, and it allows feedback from users. It's common for users to see a demonstration and say, "Oh, that isn't what I was thinking it would be like."
- It facilitates creating multiple designs for solving a problem and better mastery of things if an issue pops up.

Test Often and Keep Records

Often, project risk can be quantified numerically; for example, response times, concurrency, throughput, SLAs, uptime, and time to failover. It's important to ensure that you record these numbers and check and review them on a regular basis. Testing and retesting is an important part of this process.

As an example, say processing involves batch load operations that need to be tested for throughput and total execution time. To get the testing done early in the process, the team uses a data generator to create test data for input. The tests on this generated data end up giving great results, and the project used those results for sizing estimates for the entire project. Unknowingly to the team, the data that was generated just happened to have very low entropy, where entropy is a measure of how messy or different your data is from itself. More entropy means worse compression ratios, whereas

lower entropy means better compression ratios. In our example, the low entropy allowed the compression codec of the execution engine to read and write data faster, send data over the wire faster, and so on.

So, when the team finally gets around to testing with real data, it's a shocking surprise to find that now the job runs much slower and the data footprint was much larger than initial testing.

We can learn the following from this example:

- Test early and test often
- Keep careful records of what was tested
- Document gaps from what was tested to what will be used in production
- Try to get your test to reflect production as much as possible
- Take any measurements with a grain of salt
- Always assume your numbers are too good to be true
- Have more than one group of people review the test processes, benchmarking, and so forth

We need to take all these items into consideration when communicating numbers to people outside our project. It is difficult to try to explain why your processing is 10 times slower today than yesterday—people will tend to be unforgiving when you try to explain the gaps after the fact.

Monitoring and Alerting

We've spent a lot of time in this chapter talking about how to plan and manage risk in your architectures, but now let's talk about managing risk beyond the implementation phases and into the deployment phase. To do this, we need a way to specify and track metrics that will allow us to ensure that the system is performing as expected after it's deployed. To define these metrics, we define key performance indicators (KPIs) for each component in the architecture. For each system, your KPIs will be different, but in general, you want to start with the three key indicators defined in the book *Site Reliability Engineering* (O'Reilly) by the folks at Google:

Throughput
: This is a measure of what your system is doing and how much of it it's doing.

Latency
: This is how long it takes for your system to perform given actions.

Error rate
: This is how many errors happen with respect to given operations.

The details of defining KPIs and building the required monitoring is beyond the scope of this book, but there are many references, such as the aforementioned book, that will provide in-depth information on these tasks. The main takeaway here is that without a good way to monitor information on your systems, you're operating in the dark. Defining these KPIs and building the appropriate monitoring will play an important role in reducing risk for your project after those projects have moved beyond implementation phases and through the production deployment stages.

Communicating Risk

Talking about risks is, well, risky. If you overemphasize the reality of the risks, people viewing the project from the outside will have concerns for the project that might result in unwanted and unneeded oversight. Writing software in a large company is very much like cooking in the kitchen, and oversight can be like five cooks making one sandwich. In addition, things can become even worse if politics are involved. Politics can lead to opportunists looking to take your project away. Naysayers will have reason to debate every small topic of your design, which can result in a time-suck as well as negative perceptions of the team and can inflict moral drain on the group.

There are also risks in underplaying project risk. You might need help, and if you undersell the risk and wait too long before asking for help, you risk missed deadlines, system crashes, or lost data. At this point, the result might be that you lose the project, people can lose their jobs, and your long-term effectiveness at the company could be damaged.

So, it's easy to see that if we are communicating our risks the wrong way, we are at a minimum hurting our project, and at worst it can cost us our jobs. Communication and timing are critical when talking about risk externally. This section will attempt to go into some strategies to help both communicate risk but also communicate risk in a way that protects the progression of the project.

Collaborate and Gain Buy-In

The idea here is to reduce risk by making sure that you're collaborating closely with other people within your organization. You can do this by bouncing design ideas off people, getting others to contribute to your design, and so on. By frequently bringing in others, you can benefit in multiple ways:

Having a second (or third, or fourth) pair of eyes
: Getting additional input can often provide insight and ideas that you wouldn't have come to on your own. Regardless of your level of experience and knowledge, there are benefits to seeing a design from a different perspective.

Getting buy-in
: There is no better way to get buy-in than by adopting designs or ideas of others. When others feel like their idea is part of your design, they'll feel invested in your success.

Communicate risks openly
: If you look at risks as challenging problems to solve, not only will you get others willing to help, but you might be able to enlist upper management to help solve problems outside your control or sphere of influence.

Staying in control and demonstrating progress
: Issues and risk are present in every project, but talking about risks and communicating progress in resolving the risks will demonstrate to others your team's ability to problem solve. This can increase confidence in the ability of your team to handle future risks.

Share the Risk

Related to the suggestions we just discussed, engaging with a third party such as a consultant, vendor, or trusted member from a different department can help address risk management. Assuming that the third party is highly experienced in the problems and architectures that your project will need to address, they can help in a number of ways:

- If they have experience building this type of system somewhere else, they can help you avoid pitfalls, saving you time and money.
- They can help make decisions that are unaffected by politics.
- If they are right and make your project successful, you look good.

The first item is probably the most important. The reality is that most projects require solutions that have been built by numerous other companies. Unless you're a Google, Amazon, or Facebook, your project might be a cookie-cutter implementation for the right third-party resource. If you really want to reduce risk, getting an expert to help is often a surefire way to do it.

Using Risk as a Negotiation Tool

Finally, we'll wrap things up by noting the importance of using project risk as a tool for negotiation. This might seem like a counterintuitive concept—why and how would we want to use risk for negotiations? The fact is we can sometimes use risk to facilitate project management; for example, in negotiating for more resources or modifying scope or timelines. However, it's important to be careful when selecting which risk to use for negotiations and how to use it in negotiations.

Good candidates for negotiation are risks that are linked to a business requirement. It's very important that this is a hard link to provide a solid basis for negotiation.

A good example would be if the business is asking to store terabytes of data and be able to access any of it in milliseconds. However, the company is used to using only a system like MySQL as a datastore, which is not suited for this type of use case. Suppose that the company decides to use a NoSQL store that can handle this use case, but the team doesn't have a lot of experience with the NoSQL solution. That lack of experience is a risk, but it is a risk directly brought on by a business request.

So, with a risk that is linked to a business request, we can ask for several things in project planning, such as the following:

- Additional time in the project plan
- Additional funding for a third-party expert
- More budget for hiring, training, and so forth

Note that there are cautions with this tactic: This might set expectations higher if it's assumed that the risk will go away if management provides the requested items. Additionally, if you continually make requests, the business might doubt your ability or the ability of the team to deliver a solution. So be judicious in the requests you make and ensure that you make requests with clear insight on what you need to complete the project. In the worst case, the business cancels the project because of perceived risk of failure.

Thus, with any negotiation, know what you can ask for and what you need. Be honest as much as possible; lying in either direction will most likely catch up with you.

Summary

We've used this chapter to cover the risks you need to manage in your data projects. We then discussed how to address and minimize these risks. The types of risk we discussed were as follows:

Architectural risk
: The risk that's embodied in your technology selections, how those technologies are designed to create a solution, and so forth.

Team risk
: The risks that are represented by your project teams and external teams.

Requirements risk
: This risk can come from poorly defined requirements, requirements that are new to the team, and so forth.

We then discussed methods to address these risks, including the following:

- Creating a high-level model of your system to assign risk levels based on composites of these risk types
- Addressing technology and architectural risks, including use of abstractions, PoCs, and so forth
- Ensuring the creation of strong teams and addressing challenges working with external teams
- Ensuring the creation of solid and manageable requirements
- Using outside resources to help ensure project success

We also discussed the very important topic of communicating risk as well as using risk as a negotiating tool. Making sure that you've documented your risks, and a plan to manage and mitigate those risks will be critical to the success of your data projects.

About the Authors

Ted Malaska is Director of Enterprise Architecture at Capital One. Previously, he was Director of Engineering of Global Insights at Blizzard, helping support titles such as *World of Warcraft*, *Overwatch*, and *Hearthstone*. Ted was also a principal solutions architect at Cloudera, helping clients find success with the Hadoop ecosystem, and a lead architect at the Financial Industry Regulatory Authority (FINRA). He has also contributed code to Apache Flume, Apache Avro, Apache Yarn, Apache HDFS, Apache Spark, Apache Sqoop, and many more. Ted is a coauthor of *Hadoop Application Architectures*, a frequent speaker at many conferences, and a frequent blogger on data architectures.

Jonathan Seidman is a software engineer on the Cloud team at Cloudera. Prior to that, he was a solutions architect at Cloudera working with partners to integrate their solutions with Cloudera's software stack. Previously, he was a technical lead on the big data team at Orbitz Worldwide, helping to manage the Hadoop clusters for one of the most heavily trafficked sites on the internet. He's also a cofounder of the Chicago Hadoop User Group and Chicago Big Data, coauthor of *Hadoop Application Architectures*, technical editor for *Hadoop in Practice*, and has spoken at a number of industry conferences on Hadoop and big data.

Colophon

The animals on the cover of *Foundations for Architecting Data Solutions* are the buffalo weaver (*Dinemellia dinemelli*) and the baya weaver (*Ploceus philippinus*). Both are members of the Ploceidae family of which there are numerous species. Birds of this family are known colloquially as weavers for the way they weave together their nests from natural materials like sticks and leaf fibers.

Buffalo weavers are native to Africa and get their name from their habit of feeding on insects disturbed by the movement of the African buffalo. In addition to insects, buffalo weavers feed on fruits and seeds. They forage together in groups and are known to be highly social.

Baya weavers are found in India and southeast Asia, and are distinguished by their tube-shaped nests, which they build hanging from tree branches. Like buffalo weavers, baya weavers forage together in groups. They often feed on grains and rice, and so are classified as an agricultural pest in parts of India.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image of the buffalo weaver is from *Wood's Animate Creation* and the cover image of the baya weaver is from *Cassell's Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.