O'REILLY®

Compliments of
HITACHI
Inspire the Next

Foundations for Architecting Data Solutions

MANAGING SUCCESSFUL DATA PROJECTS

Ted Malaska & Jonathan Seidman

Change the Way the World Works

Meet Hitachi Vantara

HITACHI
Inspire the Next

Data is your greatest asset. It reveals your path to innovation and new ways for you and the world to work. At Hitachi, we've changed how we work by joining our best data technologies and people into a brand-new company: Hitachi Vantara. To elevate your innovation advantage, we are now analytics, industrial expertise, technology and outcomes rolled into one great data solutions provider. We listen. We understand. We help you drive data to outcomes that matter.

See how at HitachiVantara.com.

Foundations for Architecting Data Solutions

Managing Successful Data Projects

This Excerpt contains Chapter 1 of the book *Foundations for Architecting Data Solutions*. The complete book is available at *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

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

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.

978-1-492-03874-0

[LSI]

Table of Contents

CHAPTER 1

Key Data Project Types and Considerations

The basis for any successful data project is a clear understanding of what you're tasked to build and then understanding the major items that you need to consider in order to design a solid solution. We categorize data projects into three types that from our experience will typify many data projects. This categorization then allows us to explore the primary items we need to consider before starting on implementation. Not every project will fall neatly into one of these categories, and some projects might straddle these categories, but we feel that these project types will provide a useful framework for understanding your data use cases.

In this chapter, we begin by describing these major project types, followed by a description of the main items to consider, in general, for implementing solutions. We then take a deeper dive into these considerations for each project type.

Major Data Project Types

Let's begin by describing the three project types that we use to categorize data projects:

Data pipelines and data staging
: We can think of these as Extract, Transform, and Load (ETL)–type projects; in other words, these are projects that involve the collection, staging, storage, modeling, and so on of datasets. These are essentially projects that provide the basis for performing subsequent analysis and processing of data.

Data processing and analysis
: These are projects that end in providing some kind of actionable value. This might be the creation of reports, creation and execution of machine learning models, and so forth.

Applications
: A data framework that's meant to support live operational needs of applications; for example, the data backend for web or mobile applications.

For the rest of this chapter, we dig further into these project types by focusing on the following for each project type:

Primary considerations
: Although these three project types share many commonalities, there will also be distinctions that will affect architectural decisions and priorities. These decisions in turn will drive the rest of the project. When looking at our three project types, we begin by detailing these primary considerations for the particular project type.

Risk management
: Any data project brings with it a set of risks. We discuss possible risks associated with the project type and how to deal with these risks. In many cases, there will be multiple approaches to risk management based on specific use cases, so we need to explore this aspect along different dimensions.

Team makeup
: There are considerations regarding staffing teams to deliver the different project types. The types of skills, experience, and interests will vary with the different project types, so we provide some recommendations around building successful teams for each project type.

Security
: Another important consideration that will likely apply to all of your projects is security. This is an extensive and important topic that deserves its own book; in fact, there are useful references depending on which systems you're using. Because this is such an important topic, we won't go into detail in this book, but it's useful to enumerate the security concerns that you should keep in mind throughout your projects.

It's worth noting that security was more of an afterthought for some open source data management systems. This was because early users were more concerned with technical considerations related to the ability to store and process large volumes of data. Additionally, these systems were generally deployed on internal networks with controlled access. As enterprises began deploying these solutions, concerns about security and sensitivity of the data being stored in these systems became paramount, leading to projects and vendors working on changes and enhancements to harden these systems for enterprise use.

The following are the different dimensions that you should be considering when planning for security in your projects:

Authentication
: Ensuring that users accessing the system are who they claim to be. Any mature system should offer support for strong authentication, typically through an authentication protocol like Kerberos or Lightweight Directory Access Protocol (LDAP).

Authorization
: After you've ensured that a valid user is accessing a system, you need to determine which data they're allowed to access. A mature system will offer the ability to control access at different levels of granularity; for example, not just at a database table level, but down to column-level access. Having control over which users and groups are able to access specific data is important when building a data architecture for which the security of data is critical.

Encryption
: In addition to controlling access to data, when security is a concern, it's also important to protect that data from malicious users and intrusions. Encryption of data is a common way to help achieve this. We need to consider this from two different angles:

  - *Data at rest*. This is data that has landed in your system and is stored on disk. Many data management vendors offer solutions as part of their platform for managing this, and several third-party vendors offer solutions.
  - *Data on the wire*. This is the data that's moving through your system. Generally, vendors or projects will support this via standard encryption mechanisms such as Transport Layer Security (TLS).

Auditing
: The final dimension to security is being able to capture activity related to data in your system, including the lineage of the data, who and what is accessing the data, how it's being used, and so on. Here again, look for tools that are provided by a vendor or project to help address this.

If security is important to your use cases, the best approach will be to look for solutions or vendors that can address these four areas. This will allow you to spend more time focusing on solving your problems and less time on the details of managing the security of your data.

Data Pipelines and Data Staging

We begin our discussion with the data project type that has the widest scope of the three because it involves the path of data from external data sources to destination data sources and will provide the basis for building out the rest of your data use cases.

In this particular case, we need to design our solution with the following in mind:

- Types of queries, processing, and so on that we'll be performing against the destination data
- Any customer-facing data requirements
- Types of data collected in terms of its value

Because of the importance of this data in implementing further analysis and processing, it's crucial that we pay careful attention to modeling and storing the data in a way that facilitates further access.

Primary Considerations and Risk Management

We can break this use case into a number of primary considerations, which we discuss further in the subsequent sections:

- Consumption of source data
- Guarantees around data delivery
- Data governance
- Latency and confirmations of delivery
- Access patterns against the destination data

Let's look at these considerations and how attributes of each will affect our priorities.

Source data consumption

When we talk about data sources, we're basically talking about the things that create the data that's required for building your solutions. Sources could be anything from phones, sensors, applications, machine logs, operational and transactional databases, and so on. The source itself is mostly out of scope of your pipeline and staging use cases. In fact, you can evaluate the success of your scoping by how much of your time you spend working with the source team. The more time your data engineering team spends on source integration can often be inversely correlated to how well designed the source integration is.

There are standard approaches we can use for source data collection:

Embedded code
: This is when you provide code embedded within the source system that knows how to send required data into your data pipeline.

Agents
: This is an independent system that is close to the source and in many cases is on the same device. This is different from the embedded code example because the agents run as separate processes with no dependency concerns.

Interfaces
: This is the lightest of the options. An example would be a Representational State Transfer (REST) or WebSocket endpoint that receives data sent by the sources.

It should be noted that there are other commonly used options to perform data collection; for example:

- Third-party data integration tools, either open source or commercial
- Batch data ingest tools such as Apache Sqoop or tools provided with specific projects; for example, the Hadoop Distributed File System (HDFS) `put` command

Depending on your use case, these options can be useful in building your pipelines and worth considering. Because they're already covered in other references or vendor and project documentation, we don't cover these options further in this section.

Which approach is best is often determined by the sources of data, but in some cases multiple approaches might be suitable. The more important part can be ensuring a correct implementation, so let's discuss some considerations around these different collection types, starting with the embedded code option.

Embedded code. Consider the following guidelines when implementing embedded code as part of source data collection:

Limit implementation languages
: Don't try to support multiple programming languages; instead, implement with a single language and then use bindings for other languages. For example, consider using C, C++, or Java and then create bindings for other languages that you need to support. As an example of this, consider Kafka, which includes a Java producer and consumer as part of the core project, whereas other libraries or clients for other languages require binding to libraries that are included as part of the Kafka distribution.

Limit dependencies
: A problem with any embedded piece of code is potential library conflicts. Making efforts to limit dependencies can help mitigate this issue.

Provide visibility
: With any embedded code, there can be concerns with what is under the hood. Providing access to code—for example, by open sourcing the code or at least pro-

viding the code via a public repository—provides an easy and safe way to relieve these fears. The user can then get full view of the code to alleviate potential concerns involving things like memory usage, network usage, and so on.

Operationalizing code
: Another consideration is possible production issues with embedded code. Make sure you've taken into account things like memory leaks or performance issues and have defined a support model. Logging and instrumentation of code can help to ensure that you have the ability to debug issues when they arise.

Version management
: When code is embedded, you likely won't be able to control the scheduling of things like updates. Ensuring things like backward compatibility and well-defined versions is key.

Agents. The following are things to keep in mind when using agents in your architecture:

Deployment
: As with other components in your architecture, make sure deployment of agents is tested and repeatable. This might mean using some type of automation tool or containers.

Resource usage
: Ensure that the source systems have sufficient resources to reliably support the agent processes, including memory, CPU, and so on.

Isolation
: Even through agents run externally from the processing applications, you'll still want to protect against problems with the agent that can negatively affect data collection.

Debugging
: Again, here we want to take steps to ensure that we can debug and recover from inevitable production issues. This might mean logging, instrumentation, and so forth.

Interfaces. Following are some guidelines for using interfaces:

Versioning
: Versioning again is an issue here, although less painful than the embedded solution. Just make sure your interface has versioning as a core concept from day one.

Performance

With any source collection framework, performance and throughput are critical. Additionally, even if you design and implement code to ensure performance, you might find that source or sink implementations are suboptimal. Because you might not control this code, having a way to detect and notify when performance issues surface will be important.

Security

While in the agent and embedded models you control the code that is talking to you, in the interface model you have only the interface as a barrier to entry. The key is to keep the interface simple while still injecting security. There are a number of models for this such as using security tokens.

Risk management for data consumption

The risks that you need to worry about when building out a data source collection system include everything you would normally worry about for an externally facing API as well as concerns related to scale. Let's look at some of the major concerns that you need to be looking for.

Version management. Everyone loves a good API that just works. The problem is that we rarely can design interfaces with such foresight that they won't require incompatible changes at some point. You will want to have a robust versioning strategy and a plan for providing backward compatibility guarantees to protect against this as well as ensuring that this plan is part of your communication strategy.

Impacts from source failures. There are a number of possible failure scenarios at the source layer for which you need to plan. For example, if you have embedded code that's part of the source execution process, a failure in your code can lead to an overall failure in data collection. Additionally, even if you don't have embedded code, if there's a failure in your collection mechanism such as an agent, how is the source affected? Is there data loss? Can this affect expected uptime for the application?

The answer to this is to have options and know your sources and communicate those different options with clear understanding that failure and downtime will happen. This will allow adding any required safeguards to protect against these possible failure scenarios.

Note that failure in the data pipeline should be a rare occurrence for a well-designed and implemented pipeline, but it will inevitably happen. Because failure is inevitable, our pipelines need to have mechanisms in place to alert us when undesired things take place. Examples would be monitoring of things like throughput and alerting when metrics are seen to deviate from specific thresholds. The idea is to build the most resilient pipelines for the use case but have insight into when things go wrong.

Additionally, consider having replicated pipelines. In failure cases, if one pipeline goes down, another can take over. This is more than node failure protection; having a separate pipeline protects you from difficult-to-predict failures like badly configured deployments or a bad build getting pushed. Ideally, we should design our pipelines in a way that we can deploy them as simply as you would deploy a web application.

Protection from sources that behave poorly. When you build a data ingestion system, it's possible that sources might misuse your APIs, send too much data, and so on. Any of these actions could have negative effects on your system. As you design and implement your system, make sure to put in place mechanisms to protect against these risks. These might include considerations like the following:

Throttling
: This will limit the number of records a source can send you. As the source sends you more records, your system can increase the time to accept that data. Additionally, you might even need to send a message indicating that the source is making too many connections.

Dropping
: If your system doesn't provide guarantees, you could simply drop messages if you become overloaded or have trouble processing input data. However, opening the door to this can introduce a belief that your system is lossy and can lower the overall trust in your system. Under most circumstances, though, being lossy might be fine as long as it is communicated to the source that loss is happening in real time so that clients can take appropriate action. In short, whenever pursuing the approach of dropping data, make sure your clients have full knowledge of when and why.

Data delivery guarantees

When planning a data pipeline, there are a number of promises that you will need to give to the owners of the data you're collecting. With any data collection system, you can offer different levels of guarantees:

Best effort
: If a message is sent to you, you try to deliver it, but data loss is possible. This is suitable when you're not concerned with capturing every event. An example might be if you're performing processing on incoming data to capture aggregate metrics for which total precision isn't required.

At least once
: If a message is sent to you, you might duplicate it, but you won't lose it. This will probably be the most common use case. Although it adds some complexity over best effort, in most cases you'll probably want to ensure that all events are captured in your pipeline. Note that this might also require adding logic in your

pipeline to deduplicate data, but in most cases this is probably easier and less expensive to implement than the exactly-once option described next.

Exactly once
: If a message is received by the pipeline, you guarantee that it's processed and will never be duplicated. As we noted, this is the most expensive and technically complex option. Although many systems now promise to provide this, you should carefully consider whether this is necessary or whether you can put other mechanisms in place to account for potential duplicate records.

Again, for most use cases at least once is likely suitable given that it's often less expensive to perform deduplication after ingesting data. Regardless of the level of guarantee, you should plan for, document, and communicate this to users of the system.

Data management and governance

A robust data collection system today must have two critical features:

Data model management
: This is the ability to change or add data models.

Data regulation
: This is the ability to know everything that is being collected and the risks that might exist if that data is misused or exposed.

Data model management. You need to have mechanisms in place to capture your system's data models, and, ideally, this should mean that groups using your data pipeline don't need to engage your team to make a new data feed or change an existing data feed. An example of a system that provides an approach for this is the Confluent Schema Registry for Kafka, which allows for the storage of schemas, including multiple versions of schemas. This provides support for backward compatibility for different versions of applications that access data in the system.

Declaring a schema is only part of the problem. You might also need the following mechanisms:

Registration
: The definition of a new data feed along with its schema.

Routing
: Which data feeds should go to which topics, processing systems, and possible storage systems.

Sampling
: An extension to routing, with the added feature of removing part of the data. This is ideal for a staging environment and for testing.

Access controls
: Who will be able to see the data, both in final persisted state and off the stream.

Metadata captured
: The ability to attach metadata to fields.

Additional features that you will find in more advanced systems include the following:

Transformational logic
: The ability to transform the data in custom ways before it lands in the staging areas.

Aggregation and sessionization
: Transformational logic that understands how to perform operations on data windows.

Regulatory concerns. As more data is collected, stored, and analyzed, concern for things like data protection and privacy have grown. This, of course, means that you need to have plans in place to respond to regulations as well as protecting against external hacks, internal misuse, and so on. Part of this is making sure that you have a clear understanding and catalog of all the data you collect.

Latency and delivery confirmation

Unlike the requirements of a real-time system, a data pipeline normally gets a lot of leeway when it comes to latency and confirmations of delivery. However, this is a very important area for you to scope and define expectations. Let's define these two terms and what you'll need to establish with respect to each.

Latency. This is the time it takes from when a source publishes information until that information is accessible by a given processing layer or a given staging system. To illustrate this, we use the example of a stream-processing application. For this example, assume that we have data coming in through Kafka that is being consumed by a Flink or Spark Streaming application. That application might then be sending alerts based on the outcome of processing to downstream systems. In this case, we can quantify latency into multiple buckets, as illustrated in Figure 1-1.

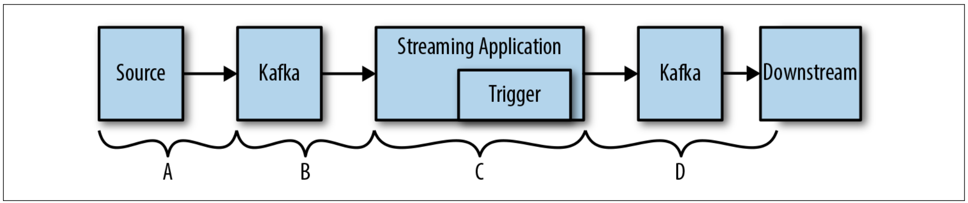

Figure 1-1. Quantifying system latency

Let's take a closer look at each bucket:

- **A**: Time to get from the source to Kafka. This can be a direct shot, in which case we are talking about the time to buffer and send over the network. However, there might be a fan-in architecture that includes load balancers, microservices, or other hops that can result in increased latency.
- **B**: Time to get through Kafka will depend on a number of things such as the Kafka configuration and consumer configuration.
- **C**: Time between when the processing engine receives the event to when it triggers on the event. With some engines like Spark Streaming, triggering happens on a time interval, whereas others like Flink can have lower latency. This will also be affected by configuration as well as use cases.
- **D**: Again, we have the time to get into Kafka and be read from Kafka. Latency here will be highly dependent on buffering and polling configurations in the producer and consumer.

Delivery confirmation. With respect to a data pipeline, delivery confirmations let the source know when the data has arrived at different stages in your pipeline and could even let the source know if the data has reached the staging area. Here are some guidelines on designing this into your system:

Do you really need confirmation?
The advantage of confirmation is that it allows the source to resend data in the event of a failure; for example, a network issue or hardware failure. Because failure is inevitable, providing confirmation will likely be suitable for most use cases. However, if this is not a requirement, you can reduce the time and complexity to implement your pipeline, so make sure to confirm whether you really need confirmation.

How to deliver the confirmation?
If you do need to provide confirmations, you need to add this to your design. This will likely include selecting software solutions that provide confirmation functionality and building logic into any custom code you implement as part of your pipeline. Because there's some complexity involved in providing delivery confirmation, as much as possible try to use existing solutions that can provide this.

Risk management for data delivery

Data delivery promises can be risky. For one, you want to be able to offer a system that gives users everything they want, but in the real world, things can and will go awry, and at some point, your guarantees might fail.

There are two suggested ways to deal with this risk. The first is to have a clean architecture that's shared with stakeholders in order to solicit input that might help to ensure stability. Additionally, having adequate metrics and logging is important to validate that the implementation has met the requirements.

Additionally, you will want a mechanism that will notify users and the source systems when guarantees are being missed. This will provide time to adjust or switch to a backup system seamlessly.

Access patterns

The last focus of the data pipeline use case is access patterns for the data. What's important to call out here are the types of access and requirements that you should plan for when defining your data pipeline.

We can break this into two groupings: access to data, and retention of data. Let's look at data access and the types of jobs that will most likely come up as priorities. This includes the following:

- Batch jobs that do large scans and aggregations
- Streaming jobs that do large scans
- Point data requests; for example, random access
- Search queries

Batch jobs with large scans. Batch jobs that scan large blocks of data are core workloads of data research and analytics. Let's run through four typical workload types that fit into this categorization to help with defining this category. To gain a solid grasp on the concept, let's discuss some real-world use cases with respect to a supermarket chain:

Analytical SQL
: This might entail using SQL to do rollups of which items are selling by zip code and date. It would be very common that reports like this would run daily and be visible to leadership within the company.

Sessionization
: Using SQL or tools like Apache Spark, we might want to *sessionize* the buying habits of our customers; for example, to better predict their individual shopping needs and patterns and be alerted to churn risks.

Model training
: A possible use case for machine learning in our supermarket example is a recommendation solution that would look at the shopping habits of our customers and provide suggestions for items based on customers' known preferences.

Scenario predictions evaluation
: In the supermarket use case, we need to define a strategy for deciding how to order the items to fill shelves in our stores. We can use historical data to know whether our purchasing model is effective.

The important point here is that we want the data for long periods of time, and we want to be able to bring a large amount of processing power to our analytics in order to produce actionable outcomes.

Streaming jobs with large scans. There are two main differences between batch and streaming workloads: the time between execution of the job, and the idea of progressive workloads. With respect to time, streaming jobs are normally thought to be in the range of millisecond to minutes, whereas batch workloads are generally minutes to hours or even days. The progressive workload is maybe the bigger difference because it is a difference in the output of the job. Let's look at our four types of jobs and see how progressiveness affects them:

Analytical SQL
: With streaming jobs, our rollups and reports will update at smaller intervals like seconds or minutes, allowing for visibility into near-real-time changes for faster reaction time. Additionally, ideally the processing expense will not be that much bigger than batch because we are working on only the newly added data and not reprocessing past data.

Sessionization
: As with the rollups, sessionization in streaming jobs also happens in smaller intervals, allowing more real-time actionable results. Consider that in our supermarket example we use sessionization to analyze the items our customers put into their cart and, by doing so, predict what they plan to make for dinner. With that knowledge, we might also be able to suggest items for dessert that are new to the customer.

Model training
: Not all models lend themselves to real-time training. However, the trained models might be ideal for real-time execution to make real-time actionable calls in our business.

Scenario predictions evaluation
: Additionally, now that we are making real-time calls with technology, we need a real-time way to evaluate those decisions to power humans to know whether corrections need to be made. As we move to more automation in terms of decision making, we need to match that with real-time evaluation.

Where the batch processing needs massive storage and massive compute, the streaming component really only needs enough compute, storage, and memory to hold windows in context.

Point requests. Until now, we have talked about access patterns that are looking at all of the data either in given tables or partitions, or through a stream. The idea with point requests is that we want to fetch a specific record or data point very fast and with high concurrency.

Think about the following use cases for our supermarket example:

- Look up the current items in stock for a given store
- Look up the current location of items being shipped to your store
- Look up the complete supply chain for items in your store
- Look up events over time and be able to scan forward and backward in time to investigate how those events affect things like sales

When we explore storage in a later chapter, we'll go through indexable storage solutions that will be optimal for these use cases. For now, just be aware that this category is about items or events in time related to an entity and being able to fetch that information in real time at scale.

Searchable access. The last access pattern that is very common is searchable access to data. In this use case, access patterns need to be fast while also allowing for flexibility in queries. Thinking about our supermarket use case, you might want to quickly learn what types of vegetables are in stock, or maybe look at all shipments from a specific farm because there was a contamination.

Risk management for access patterns

In previous sections, the focus was on data sources and creating data pipelines to make the data available in a system. In this section, we talk about making data available to people who want to use it, which means a completely different group of stakeholders. Another consideration is that access pattern requirements might change more rapidly simply because of the larger user base trying to extract value from the data. This is why we've organized our discussion of access patterns into four key offerings. If you can find a small set of core access patterns and make them work at scale, you can extend them for many use cases.

With that being said, doing the following will help ensure that you manage risks associated with access patterns:

- Help ensure that users are using the right access patterns for their use case.

- Make sure your storage systems are available and resilient.
- Verify that your offering is meeting your users' needs. If they are copying your data into another system to do work, that could be a sign that your offering is not good enough.

Pipeline and Staging Team Makeup

Now that we have walked through primary considerations and risks for pipeline and staging use cases, you might have begun to get an idea of the types of people that you want on these teams. The following are some job roles to consider:

Service and support engineers
These are engineers who are tasked with working with users, including stakeholders of source data systems, users accessing data in the system, and so on. These engineers are trained to do the following:

- Optimally use the system
- Look for good use cases for users of the system
- Work with the team to address issues being faced by users
- Advocate for users
- Help users be successful

System engineers/administrators
These are engineers who are obsessed with uptime, latency, efficiency, failure, and data integrity. They don't need to care much for the real use cases for the system; rather, they focus more on the reliability of the system and its performance.

Data engineers
These are engineers who know the data and the types of storage systems within your system. Their main job is to make sure that the data offerings are being used correctly and that the appropriate big data solutions are being used for the right thing. These are experts in data storage and data processing.

Data architects
These are experts in data modeling who can define the structures that can define how data in the system should be structured. In some cases, this role might be assumed by the data engineers on the team.

Data Processing and Analysis

These are the projects for which we use the data populated by the data pipeline and staging projects and then transform and analyze the data to derive value. We've already talked a little about transformation and value creation in the previous discussion. However, that discussion mainly focused on preparing the data for use in our data processing and analysis applications. In this section, we examine the use cases that perform those transformations and analysis on the data to derive value.

Primary Considerations and Risk Management

We focus on the following high-level items when we're evaluating this type of use case:

- Defining the problems that we're trying to solve
- Implementing and operationalizing the processing and analysis to solve the problems we've identified

In short, what do we want to do, how do we get there, how do we make it repeatable, and how do we quantify its value when we get there?

Defining the problems to be solved

In many cases, your business will already know many of the questions that need to be answered to provide value. Often, though, determining the appropriate questions to ask and which are the correct problems to solve for your business can be a challenge. This is especially true when there are many possible questions to pursue, some of which are potentially impactful, others cool and interesting. Other times the questions we should be asking aren't even obvious.

So, the first focus here is really not about the data or what we want to do with the data, but about the value we want to derive from the data; what will make an impact to our business, and what can be actionable? Determining the answers to these questions requires talking to stakeholders—the users and customers of your systems. Often, the challenges here can be getting the time and attention of these stakeholders and, unfortunately, it's often the case that even they won't know what they really need.

The following are some good questions to ask yourself and users during this process:

- What are the most important things to capture in terms of trackable metrics?
- What are things that affect customer engagement?
- Is there a gap in the company's offerings?

- Are there active pain points?

A good next step is to try to further define our objectives in solving the problem. This will generally mean defining specific metrics, numbers, visualizations, and so on that we can use to help evaluate approaches to solving the problem and determine whether and how other potential issues might relate to the problem we're addressing.

Let's take this idea and apply it to our supermarket example. Suppose that our problem is a concern that shelves are not fully stocked with the proper products. We could create visualizations to help illustrate this, such as the following:

Products stocked over time
: A time-based chart that shows the average, max, min, top 10%, and bottom 10% of inventory for each product

Products out of stock
: A time-based chart illustrating times when each item is unstocked

These charts will help us drill into additional visualizations that can help us define the impact of this problem; for example:

Substitute purchase patterns
: During times when a product is out of stock, do customers select a different product?

Delivery supply chain
: For products that are understocked, was the problem caused by delivery delays?

Region or store variations
: Are there differences in stock levels across regions or stores?

Customer impact
: For customers who are normal purchasers of the understocked items, do we see a change in spending between time periods when inventory levels are acceptable and when the item is out of stock?

The idea is to provide context that will help illustrate the problem, provide scope around the problem, and show potential impact of the problem. It's possible that this exercise will provide enough information to drive actionable decisions. This information will also play a role in determining how important this problem is to solve versus other problems facing the company.

Risk management for problem definition

When identifying problems that need to be solved, two important considerations can help you manage risk and will affect further efforts:

Get many viewpoints

Make sure that you get input from multiple sources, not just one or two. Examples would be just talking to top management or a team directly affected by the problem. The issue is that different groups might be too close or too far from the problem to be able to see it from all angles. Having the additional perspectives can help you in terms of how to later quantify and solve the problem.

Build trust

You want to be seen as a collaborator, and not somebody trying to find shortcomings or issues within the organization. Be sure that you work closely with stakeholders to gain trust.

The challenge at this stage is confirming that you're defining the correct problem and capturing it correctly. An effective way to address this concern is to communicate about the problem you're trying to solve often and openly. This might seem like common sense but can often help avoid the perception of placing blame or responsibility on parts of the organization. It can sometimes be the case that putting the focus on a specific problem can have the appearance of putting people or groups in your organization in a negative light.

The way around these political traps can be to find cross-departmental contacts that understand the iterative process of problem definition and then communicating often with these folks to reduce any nasty surprises.

Implementing and operationalizing solutions

Now that we have a handle on the problem and we've explored solutions, we're ready to move on to implementing these solutions. While keeping in mind the original problem we're trying solve, it's important to remain flexible and adapt to new information that might affect our solution.

A couple of important considerations to be mindful of during the implementation phase are to focus on building the right components in a robust way to ensure that you're not just creating one-off solutions, and to operationalize solutions.

Building a robust solution. A common trap is to build out systems that solve a single problem. A better approach is to build systems that can provide a platform for solving multiple problems. Being able to get results quickly and correctly is good, but even better is to be able to find all (or as many as possible) of the answers quickly and correctly.

You want to ensure that you define a clear path to value. A useful way of doing this can be to create a block diagram, such as that depicted in Figure 1-2, that goes from top to bottom to help visualize your path. On the uppermost side you put all the data sources, and as you move from top to the bottom, you get closer and closer to value and actionable products.

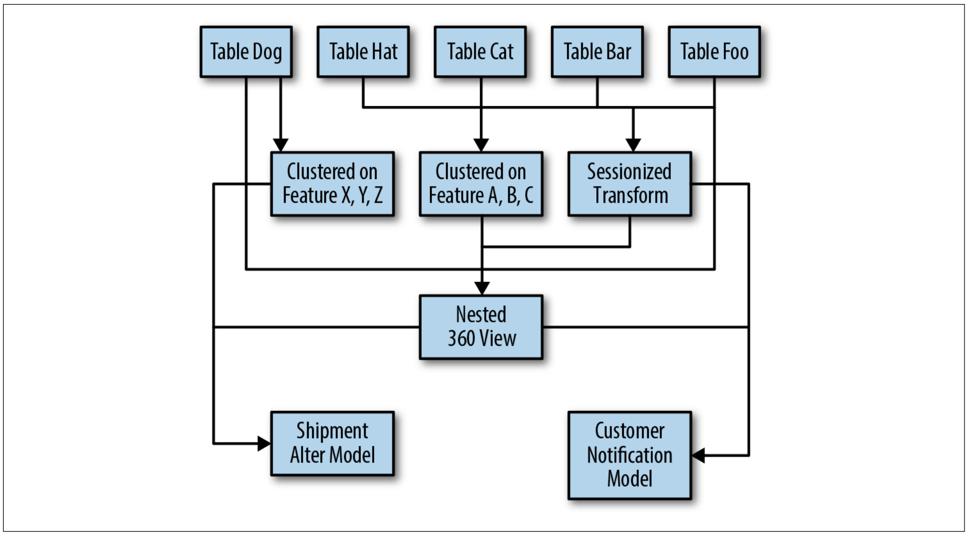

Figure 1-2. Breaking a system into blocks

Think of this as building blocks. You want to build blocks like the Nested 360 view that can be useful for addressing many problems; not just the current ones, but also future needs.

The risk to building too many blocks that have single uses is that every block must be maintained. At some point, there can be value in consolidating blocks; the goal should be to streamline your paths to value.

On the other hand, if you try to build a system to solve every problem, you might end up with a system that's not well suited to solve any problem. It's always good to start small and make sure you have a clear understanding of the business requirements.

Operationalizing solutions. A common trap is trusting the same people who find the problems or solve the problems to operationalize the solutions, because these different activities require completely different skill sets. You will want a good pipeline and communication between these two groups. It's also important that you make the same commitment to operationalizing solutions as to actually finding the solutions.

Data Processing and Analytics Team Makeup

Unlike the pipeline and staging team, teams that can successfully use data to provide value need to be more focused on problem discovery, working across teams, and finding paths to solutions. This means that we need a different mix of people:

Problem seekers
: These are people who are able to earn trust with different groups in your company and are adept at identifying and quantifying problems. Think of them as treasure hunters who need to build alliances and search through the weeds of everyday business to find problems to solve that will have a high impact. These folks could have different roles within your team such as project managers, product managers, and technical leads, but could also be the individual developers and analysts on a team.

Architects
: A requirement for a successful problem-solving group is to pick not only the right problems to solve, but also the correct order. This applies very much to the block building idea; we want to pick the problems to build with the least effort and that include reusable parts.

The brains
: These are the data scientists and analysts who will help come up with solutions.

The engineers
: You'll need a group of engineers who know how to work with all of the aforementioned parties and how to take their work and productionize it.

Solution communication experts
: As we just mentioned, finding a solution can be perceived as pointing fingers. Additionally, a solution with no buy-in will never see real-life implementation. It is these communication experts who are able to evangelize the solutions so that they get to reach their full potential. This will probably be the project managers, product managers, or technical leads on a project.

Application Development

So, up until this point, we have talked about data pipelines that land data in staging areas, then we talked about projects focused on exploring data and gaining value from that data. Whereas those two use case categories are more about data gathering and learning, this final use case category is about deploying applications that use our data to provide some service to users, either internal or external. A good example of this might be a website that relies on the data to drive functionality. The assumption here is that the data is being used to drive applications that support large numbers of users, requiring scalability, reliability, and availability.

Primary Considerations and Risk Management

In this use case, key considerations to help ensure success include the following:

Latency and throughput
: How long does it take to execute an operation, and how many operations can the system handle per second?

State locality and consistency
: If your system is available in more than one region, how is replication handled? Is it siloed, eventually replicated, or strongly locked?

Availability of systems
: What are the failure and recovery characteristics of the system?

Latency and throughput

When building out requirements for an application, a good place to begin is learning what data you need to have saved and how you're going to interact with that data. For example, this interaction might be number of operations: inserts, updates, multiple-row transactions, and so on.

There are number of potential concerns we need to be considering for every data interaction in our design:

Race conditions. For example, if two clients update the same row at the same time, who wins? There are ways to address this, with the following being a few examples:

Last one wins
: In this design, it doesn't matter, and they just randomly overwrite each other.

Transactional locking
: This mean that there is a lock either at a data-store level or the server level before mutation is allowed. A common example is making a mutation only if another condition is true; for example, updating column foo to 42 if column bar equals 100.

Asynchronous versus synchronous operations. In systems that require very low latency, it might be unrealistic to save in real time. There might be a desire to have state in server or client memory and persist to the final store asynchronously. When looking at the use case requirements for this, you need to consider the following:

Speed versus truth
: With an asynchronous model, there is a short window during which there is a chance for data loss. You need to evaluate how important that is compared to latency.

Performance consistency. Does the tested performance hold? There are many questions to be asked here:

- What if the insertion order changes?
- Will the performance change as the data scales?
- Does performance change as the storage solution grows?
- If there a difference between generated data and real data?
- How will performance be affected by maintenance work?

Risk management for latency

The best way to mitigate risk in relation to performance is to test early and often and communicate always. Everything around performance should be monitored like crazy. You need to fully document every connection, be it internal or external. Additionally, you should question results often. Have more than one team test the results.

Lastly, make sure you use interfaces in your design so that you can swap out implementations for other implementations. The odds are high that you will think of a better strategy after you build the first solution. Give yourself some room to implement those changes without having to rewrite your entire system.

State locality. There are a number of places where state can exist. In a distributed system, we can highlight four major locations to hold state:

Clients
: The client-side interface the user is using

Server
: The server or servers that the client accesses

Datacenter
: Persistence with the local datacenter housing the application servers

Multidatacenter
: Replicated persistence across datacenters

When looking at use case goals and requirements, there are things to consider for each of these scenarios.

Client. Caching data in the client and allowing client-side mutation provides high performance and scalability. However, there a couple potential problems with client-side state:

Ephemeral
: The client can die at any time, causing data loss before data reaches the server.

Trust
: The client runs on potentially untrusted hosts, with untrusted users, which can cause issues where client trust is important.

Server. Although on the server side we should not be concerned with trust issues, there can still be concern with ephemeral data being lost. Another potential concern with the server is user partitioning. Whereas a client is probably partitioned one per user, a server is partitioned by groups of users. In some cases, user partitioning is based on a clear strategy; for example, with a game application that needs to maintain user state within a single server. In many cases, though, user placement is more random; for example, a web application for which user placement is based on a load balancer.

In use cases that might require multiserver state management, we might need to consider datacenter persistence.

Datacenter. This layer is where you'd see state persistence for common technologies like NoSQL systems, relational databases, and distributed caches. The goal with this layer is to store all the state needed for this local region and the server and clients within this region.

The data in this datacenter will most likely be replicated and protected from failure of one or more nodes, although that doesn't mean it is perfectly protected. It's not unheard of for regions to go down. If this is a concern in your use case, this is where a multidatacenter approach might be required.

Multidatacenter. There are several models for the multidatacenter scenario; which ones you use will depend on the requirements for your use case. Let's look at a couple that might apply to you:

Replication for disaster recovery
: In this example, we use a multidatacenter configuration to provide protection against data loss, for example, if an entire datacenter becomes unavailable. This is normally an asynchronous or batch process in which data becomes eventually consistent across datacenters. We're not looking for completely consistent state locally, but only that it is moving in that direction. This won't protect against all data loss, but it does protect against most of it.

  A concern with using replication alone is what happens if items are being mutated in multiple regions at the same time? We don't have globally consistent state, so there's no single source of truth.

Locking

Global locking is one solution to ensuring consistency across regions in cases of mutated data. The idea here is a resource is globally locked to a region. No one can mutate that resource until they get the lock. There are a number of designs to support this:

Locking mutation to a client

Every client will need to obtain a lock to change the given record.

Locking mutation to a datacenter

All mutations have to go through a given datacenter first. This requires less locking than the client scenario but will result in higher network latencies for clients that happen to be outside the region.

Locking at the record level

This is essentially a quorum architecture in which we have an odd number of locations storing state, and as long as a majority agree with the state, the changes are accepted.

A factor to keep in mind is that generally the data within a datacenter is limited to the users within the given region. If your application requires interaction between clients in different regions, you need to think about how different regions will share their data.

Risk management for locality

It's important to have a strategy about state early in your project planning; what are the requirements and goals of your project? How do decisions about state affect the user? These are difficult things to change after a project is underway.

After you have made your decision, you should fully document these choices and all effects on the user need to be enumerated. This documentation needs to be in a format that's accessible to all users and that clearly communicates the impact.

Availability

Availability of systems is of course a critical concern, but also challenging. Multiple things can affect the uptime of your systems, including the following:

Human errors

People make mistakes; a bad configuration change, a deployment of a wrong version of code, and so on.

Upgrades

Some upgrades will require the system to be restarted. Even in the case of rolling restarts, there are parts of the system that need to be unavailable for some period.

Failures
: Hardware failures are inevitable, regardless of whether you're running systems on-premises or in the cloud.

Attacks
: Malicious attacks are also something for which you must plan, given that they have the potential to affect the availability of your systems.

You'll need to be able to define the failure and recovery scenarios that are acceptable for your given service-level agreements (SLAs). Here are some examples of failure points and recovery plans:

Server failover
: In the case of an active server failing, you need to be able to failover to another server. If your state is at the datacenter, a single server failure should have little or no impact.

Nonreplicated cache failover
: There are some designs for which a cache is populated with a partition of the state needed to support your use case. When that cache is destroyed because of a failure, the data might be persistent in other stores; however, it takes time to restore that cache.

Eventually replicated datacenter failover
: A common way to do replication over a wide-area network is eventual replication. In case of failure, requests can switch to different datacenters and continue operating. However, there are two main problems with multidatacenter failover: first, you have a strong chance of losing data that is about to be in flight; this window will hopefully be small and it will depend on your throughput and your latency. The second problem is determining how to manage writes; for example, choosing between having a leader that manages writes or just accepting writes to any datacenter.

Risk management for availability

A good strategy to address potential scenarios that can affect your availability is to intentionally introduce failure into your system on a regular basis; for example, using a tool like Chaos Monkey, developed by Netflix.

The results of these failure tests should be used to define the impact, recovery plans, and steps to make the failure less impactful in the future. Publishing this as part of the system is not a bad idea and will provide users of the system a better window into real expectations.

Additionally, if you do this properly, you can develop a culture that not only is thinking failure first but is motivated to reduce the impacts of failure.

Application Development Team Makeup

Unlike our other two use cases, application development is about direct user impact, consistency, behavior, and efficiency of data movement. Although you might have some of the same resources you had on the pipeline use case, there are some notable differences. Specifically, consider the following types of resources as part of your application development team:

Site reliability engineers (SREs)
: These are engineers who are dedicated to ensuring the reliability and scalability of applications deployed to production. These resources will be critical to the success of the deployment of your applications.

Database engineers
: These probably won't be traditional database developers and architects, but rather people who have a deep understanding of modern distributed data storage and processing systems. These are the folks who are going to make sure that reading, writing, and transactions are executing at the high and consistent speed you need to make your product viable.

Summary

This has been a long chapter with a lot of material, but it's important that you have a good understanding of your projects and go through a thorough planning process before moving forward with implementation. To facilitate this process, we've broken data projects into the three most common categories:

Data pipelines and staging
: These are projects that involve bringing source data into your systems and preparing them for further processing. These projects will provide the basis for all other types of projects in your organization, so it's critically important to pay careful attention when planning these projects.

Data processing and analysis
: After the data is available, these are projects that seek to gain actionable insights from your data by performing processing and analysis of the data. These projects might be ad hoc explorations performed by analysts or full-blown projects that drive reports and dashboards for business users.

Applications
: These are user-facing applications that provide services and value to users, either internal or external. These projects will generally rely on successful implementations and deployments of the previous two project types.

For each of these project types, we discussed considerations that should drive your project planning and development. We broke these considerations into three types:

Primary considerations
: Unique considerations that you should include in your planning for each project type

Risk
: Likely project risks for which you should plan, and information to help you mitigate these risks

Project teams
: Roles that you should consider when forming teams to deliver each project type

The guidelines in this chapter are based on our experience working on multiple projects across different companies. They should help to ensure the success of your data projects. We'll also cover some of these topics in more detail in the next chapters.

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.