Fundamentals of Data engineering Chapter 4

Favorite	☐
Archived	☐
Fleeting	☐
Area/Resource	💽 Data engineering
Project	
Type	
Review Date	
Image	
URL	
Created	@March 13, 2023 12:07 PM
Updated	@March 29, 2023 10:28 PM
Root Area	https://www.notion.so/59a9ea296b924e64af4b69632f2dc92f
Project Area	
Updated (short)	03/29/2023
Pulls	
Resource Pulls	
Project Archived	
URL Base	
🍳 Recipe Divider	🥗🥗🥗 RECIPE BOOK PROPERTIES 🥗🥗🥗
🍳 Recipe Tags	
📚 Book Divider	📚📚📚 BOOK TRACKER PROPERTIES 📚📚📚
📚 Author	

📚 Date Started	
📚 Date Finished	
📚 Book Status	
📚 Rating	

💡 This chapter is concerned with choosing technologies across the data engineering lifecycle. It is easy to get caught up in chasing bleeding-edge technology while losing sight of it's core purpose which is designing robust and reliable systems to carry data through the full lifeczcle and serve it according to the needs of end users.

Architecture first design

💡 We should never design our architecture based on tools but based on the value added to the end user. The Architecture is the what, why and when whereas the tools are the how.

The key considerations when choosing tools are:

- Team size and capabilities
- Speed to market
- Interoperability
- Cost optimization and business value
- Today versus the future: immutable versus transitory technologies.
- Location
- Build versus buy
- Monolith versus modular
- Serverless versus servers

- Optimization, performance the benchmark wars
- The undercurrents of the data engineering lifecycle

Team size and capabilities

💡 The team size and their capabilities is important in determining the complexity of the tools you adopt to produce the architecture you designed.

Small teams

💡 Small teams should put off as much technical complexity and focus on providing value by using managed solutions and SaaS. Your limited bandwith should be dedicated on adding value through their tools not managing them.

Large teams

💡 Large teams with more in dept and diverse skill sets can benefit from more complex tools as they have more bandwith to handle the effects of managing them.

Speed to Market

💡 Speed to market always wins in technology. Choosing the right technologies that help you deliver features and data faster while maintaining high quality standards and security is crucial. It also means working in a tight feedback loop of launching, larning, iterating and making improvements. Deliver value early and often. Choose tools that help you move quikcly, reliably, safely and securely.

Interoperability

✏️ Interoperability describes how various technologies or systems connect, exchange information, and interact.

💡 How easily do tools interact with the rest of your tools. There is often a spectrum of difficulty ranging from seamless to time-intensive. Do you need a lot of effort to integrate technologies? You should always be aware of how simple it is to connect your various technologies across the data engineering lifecycle.

💡 You should design for modularity and giving yourself the ability to easily swap out technologies as new practices and alternatives evolve.

Cost optimization and business value

💡 Your organization expects a positive ROI from your data projects, so you must understand the basic costs you can control.

Total Cost of Ownership (TCO)

✏️ TCO is the total estimated cost of an initiative, including the direct and indirect costs of products and services utilized.

Direct costs

Direct costs are directly attributed to an initiative. Such as the salaries of the workers involved or the bills for the services consumed.

Indirect costs

Indirect costs or overhead are independent of the initiative and must be paid regardless of where they're attributed.

How something is purchased impacts the way costs are accounted form. Expenses fall into two big groups: Operating expenses, and captial expenses.

Capital expenses, CAPEX

Capital expenses: Require an upfront investment. Payment is required today. Before the cloud existed most expenses were capex for data processing as you would typically purchases hardware and software up front.

Operational expenses, OPEX

Operational expenses: Gradual expenses that are spread out over time. Whereas capex is long-term focused, opex is short term. Opex can be pay as you go and offers a lot of flexibility. Opex is closer to a direct cost making it easier to attribute to a data project.

Witht eh advent of the cloud, data platform services allow engineers to pay on a consumption-based model. Opex allows for far greater ability for engineering teams to choose their software and hardware. Given the flexibility engineers should take an opex-first approach centered on the cloud and flexible pay-as-you-go tech.

Total Opportunity Cost of Ownership (TOCO)

✏️ Total opportunity cost of ownership is the cost of lost opportunities that we incur in choosing a technology, architecture of process.

- If you choose a tool A you are forgoing the benefits of using tool B
    - You’re commited to tool A and everything it entails
    - What happens if tool A becomes obsolete?

FinOps

💡 The goal of FinOPs is to fully operationalize financial accountability and business value by applying the DevOps like pracitces of monitoring and dynamically adjusting systems.

> If if seems that FinOps is about saving money, then think again. FinOps is about making money. Cloud spend can drive more revenue, signal customer base growth, enable more product and feature release velocity, or even help shut down a data center.

Today versus the future: Immutable versus transitory technologies

💡 You should focus on the present and near future when choosing technologies but also in a way that allows you to handle future unknowns and evolution. Ask yourself where are you today and where would you like to be in the future? These answers should inform decisions about your architecture. This is done by **understanding what is likely to change and what tends to stay the same.**

Immutable technologies

✏️ Immutable technologies are components, languages or paradigms that have stood the test of time. Such as S3 object storage. These technologies benefit from the Lindy effect which state that the longer a technology has been established the longer it will be used.

Transitory technologies

✏️ Transitory technologies are those that come and go. They typically come with a lot of hype, followed by meteoric growth and a slow descent into obscurity.

💡 Identify immutable technologies fromt transitory ones every two years. Once you identify immutable technologies use them as your base and build transitory tools around the immutables. You should also consider how easy it is to transition from a chosen technology.

Location

- Premises
- Cloud
- MultiCloud
- Hybrid

Premises

💡 Most startups are born in the cloud, and on premises is still the default for established companies. When migrating to the cloud it is important that companies have an understanding of the opex first approach of cloud pricing.

Cloud

💡 Users are able to dynamically scale resources that were inconcivable with on-premises servers. This dynamic scaling makes cloud models extremely appealing to agile teams.

- IaaS
- Paas
- SaaS

IaaS

💡 Infrastructure as a service are products such as VMs and virtual disks that are essentially rented slices of hardware. Slowly we have seen a shift towards platform as a service (PaaS)

PaaS

💡 PaaS includes IaaS products but adds more sophisticated managed services to support applications. They allow engineers to ignore the operational details of managing individual machines and deploying frameworks across distributed systems. Providing turneky access to complex, autoscaling systems with minimal operational overhead.

- DataBricks

SaaS

💡 SaaS offerings move one step up the ladder of abstraction. They provide a fully functional enterprise software platform with little operational management.

- SalesForce
- Microsoft 265

💡 Enterprises that migrate to the cloud often make major deployoment errors by not appropriately adapting their pracitces to the cloud pricing model.

Cloud economics

💡 Cloud services are similar to financial derivatives. Cloud providers use virtualization to sell slices of hardware, but also sell these pieces with varying technical characteristics and risks attached. There are massive opportunities for optimization and scaling by understanding cloud pricing.

Cloud providers on risk

💡 Cloud vendors deal in risk. They offer services that have certain technical characteristics that you anticipate you will need. For example archive storage. The risk is that while storage is cheap, if you ever need the data you will pay a high cost to retrieve it.

💡 Rather than charging for CPU cores, or other features, cloud providers charge based on characteristics such as durability, reliability, longevity and predictability. Compute platforms offer workloads that are ephemeral or can be interrupted where capacity is needed elsewhere.

Curse of familiarity

💡 Many products are intentionally designed to look like something familiar to faciliate ease of user and accelerate adoption. But any new technology product has subtleties and wrinkles that users must learn to identify, accomodate and optimize.

Increase business value

💡 We should aim to increase business value by leveraging the dynamic value of the cloud.

Data gravity

💡 The concept that cloud providers will lock you in through high egress fees, and other fees that make it expensive to swap providers.

Location Part 2

Hybrid cloud

💡 As companies may believe they have achieved operational excellence is some areas of their own prem solutions they may only want to migrate certain workloads to the cloud.

Multicloud

💡 Deploying workloads to multiple public clouds. Data intensive applications make want to use multi cloud solutions where netowrk latency, and bandwith limitations hamper performance.

Disadvantages

💡 The complexity of managing the overheads relating to a multi cloud solutions usually makes it this a bad solution. A new generation of cloud of clouds toos aims to reduce this complexity by offering services across clouds, seamlessly replicating data between clouds or managing workloads across clouds from a single work pane. Snowflake makes use of this, it is an evolving trend worth paying attention to.

Advice

💡 Have an escape plan. Preparing to move away from your current solution will make you more agile.

Managing your own hardware

💡 Very large companies may have specific needs and the economies of scale at which managing your own hardware and network connections make sense. This usually requires massive workloads.

- Cloudflare
- Dropbox
- Netflix and their custom CDN

Build versus Buy

💡 A major concept in Data Engineering is to invest in building and customizing when doing so will provide a competitive advantage for your business. Otherwise stand on the shoulders of giants and use what is already available.

Type A engineer

💡 A stands for abstraction. This engineer will avoid undifferentiated heavy lifting, keeping data architecture as abstract and straightforward as possible and not reinventing the wheel. They manage the data engineering lifecycle by using entirely off the shelf products.

Type B engineer

💡 B stands for build. Type B data engineers build data tools and systems that scale and leverage a company’s core competency and competitive advantage. These types of engineers are usually found at a company in stage 2 and 3 (scaling and leading with data).

💡 This distinction is important because it points out that whenever possible one should lean towards type A behavior and embrace abstraction. However, when a competitive advantage can be provided lean towards type B behavior.

Open Source software

💡 Any commitment to an OSS in a data engineering lifecycle should be properly scrutinized, as it will determine the usability within a project.

There are two types of open source software:

- Community managed OSS
- Commercial OSS

Community managed OSS

💡 A strong community and vibrant user base can underpin a successful OSS project. The following should be taken into consideration with a community managed OSS project:

- Mindshare
    - Avoid adopting projects that don’t have traction and popularity. Look at the number of GutHub stars, forks and commit volume and recency.
- Maturity
- Troubleshooting
- Project management
- Team
- Community management
- Roadmap

Commercial OSS

> 💡 Sometimes an OSS will have drawbacks such as needing to host and maintain the solution in your environment. Commercial vendors will try and solve this management headache by hosting and managing the OSS solutions for you, typically as a cloud SaaS offering. Examples include: Databricks (spark), Confluent (Kafka), DBT labs (dbt).

Factors to consider:

- Value
    - Is the value of the managed solution actually better or just a bunch of bells and whistles
- Delivery model
    - How do you access the service? API, download, Web UI?
- Support
- Releases
- Sales cycles
- Community support

Proprietary walled gardens

💡 Many companies sell closed source products. These include independent companies and cloud platform offerings.

Independent offerings

💡 Caused by a surge in VC funding many companies jumped on the data hype train and created solutions that may not be necessary. However there is some value to be found.

Cloud platform proprietary service offerings

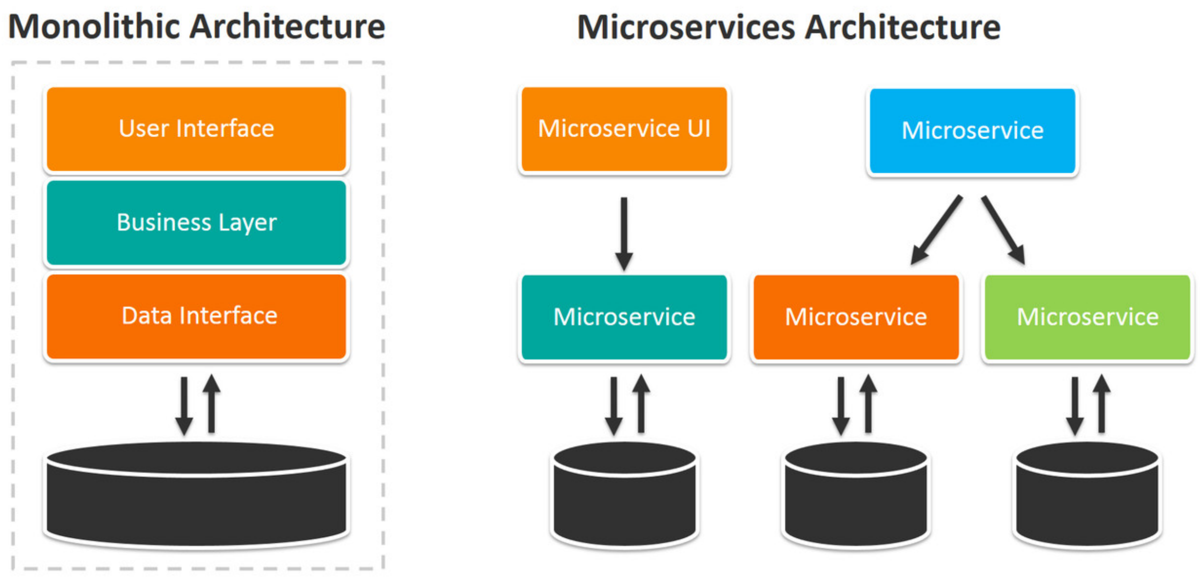
💡 Cloud vendors develop and sell their proprietary services for storage, databases and more. Many of these solutions are internal tools used by respective sibling companies. Cloud vendors will often bundle their products to work well together. Each cloud can create stickiness with its user base by creating a strong integrated ecosystem.

Advice

💡 Build versus buy comes back to knowing your competitive advantages and when an internally managed solution best fits that use case. Often it is best to stick to OSS and COSS solutions, as well as services offered by companies. However there is also value in developing your internal teams skill set in managing sophisticated systems.

Monolith versus Modular

💡 Monolith versus Modular is a longtime debate in software architecture. Monolithic systems are self contained, often performing multiple functions under a single system. It favors the simplicity of having everything in one place. It's easier to reason about a single entity and you can move faster because there are fewer moving parts. The modular camp leans toward decoupled, best-of-breed technologies performing tasks at which they are uniquely great.



Monolith

💡 Monoliths are brittle and easy to break due to their large moving parts. Changes in one part can have unforeseen consequences on other parts. Additionally pipelines that take a long time to run may have to be completely restarted if something breaks in it. This downside is to be weighed with the upside of a more simple code base that may use a single technology and one principal programming language. If you want simplicity monoliths are a good way to go.

Multitenancy

✏️ Multitenancy is when multiple applications operate in the same single environment.

💡 Multitenancy in a monolithic environment can be a major problem as it can be challenging to isolate the workloads of multiple users. One user taking up a lot of CPU can impact the performance of other users.

Modularity

💡 This returns us to the Bezos API mandate which enabled the decoupling between applications which allowed for refactoring and decomposition. He also implemented the two pizza rule which stated that no team should be so large that two pizzas can't feed the whole group. This limited the complexity of a team's domain of responsibility.

Refactoring

✏️ This refers to making changes to your code without impacting it's external behavior. This will include changing the logic of the code, changing variable names to improve quality, re usability and extensibility of the code.

Decomposition

✏️ This refers to breaking down a large application into a smaller one with more modular components. This makes an app easier to understand and maintain and enables the individual testing of components.

💡 The ability to swap out tools as technology changes is invaluable. Data modularity is a more powerful paradigm than monolithic data engineering. As it enables to teams to be agile and adopt new tools. The con of modularity however is that there is more to reason about there is more to handle and manage.

Distributed Monolith Pattern

💡 This is a distributed architecture that still suffers from many limitations involved in monolithic architecture. The basic idea is that one runs a distributed system with different services to perform different tasks. Still, services and nodes share a common set of dependencies or a common codebase.

Airflow

💡 Airflow suffers from this problem. While it utilize's a highly decoupled and asynchronous architecture, every services runs the same codebase with the same dependencies. Any executor can execute any task, so a client library for a single task must be installed on the whole cluster. This will lead to dependency conflicts. A solution for this is the ephemeral infrastructure in a cloud setting. Each job gets its own temporary server or cluster installed with dependencies. Each cluster remains highly monolithic but separating jobs dramatically reduces conflicts.

Serverless versus Servers

💡 It is important to consider the drawbacks and pros of deploying jobs in serverless offerings or managing your own servers.

Serverless

💡 Serverless enables pay as you go consumption pricing models and has many flavors such as FaaS (Function as a Service). It is important to understand the pricing impact of the different solutions. One hand it could be cheaper to not pay for an entire server at all times, however serverless functions suffer from an inherent overhead innefficiency. Handling one event per function call at a high event rate can be catastrophically expensive. Especially when simpler approaches like multithreading or multiprocessing are great alternatives.

💡 It is important to monitor and model. Monitor to determine cost per event and model using this cost per event to determine overall costs as event rates grow.

Containers

💡 Containers which are referred to as lightweight virtual machines play a large role in both serverless and microservices. They provide some of the principle benefits of virtualization without the overhead of carrying around an entire operating system kernel. Containers provide a partial solution to the distributed monolith pattern as hadoop can contain containers allowing each job to have it's own isolated dependencies.

⚠️ Containers do not provide the same security and isolation as full VMs. Container escape, a class of exploits whereby code in a container gains privileges outside the container at the OS level - is common enough to be considered a risk for multi tenancy.

How to evaluate server versus serverless

💡 Serverless makes less sense when the usage and cost exceed the ongoing cost of running and maintaining a server.

Expect servers to fail

💡 View servers as ephemeral resources that you can create as needed and then delete. If your application requires specific code to be installed on the server, use a boot script or build an image. Deploy code to the server through a CI/CD pipeline.

Use clusters and autoscaling

Treat your infrastructure as code

💡 Automation doesn't apply to just servers and should extend to your infrastructure whenever possible.

Use containers

💡 For sophisticated or heavy duty workloads with complex installed dependencies, consider using containers on either a single server or K8s.

Advice

Workload size and complexity

💡 Serverless works best for simple, discrete tasks and workloads. It’s not as suitable if you have many moving parts or require a lot of compute or memory horsepower.

Execution frequency and duration

💡 How many requests per second will your serverless application process? How long will each request take to process? Cloud serverless platforms have limits on execution frequency, concurrency and duration.

Requests and networking

💡 Serverless platforms often utilize some form of simplified networking and don’t support all cloud virtual networking features, such as VPC’s and firewalls.

Languages

Runtime limitations

💡 You are limited to a specific runtime image

Cost

Optimization, Performance and the Benchmark wars

💡 In the data space many vendors will use nonsensical benchmarks comparing services that are optimized for completely different use cases.

Nonsensical cost comparisons

💡 Comparing ephemeral and non-ephemeral systems on a cost-per-second basis is nonsensical, but we see this all the time in benchmarks.

Undercurrents and their impacts on Choosing technologies

💡 Whatever technology you choose make sure to understand how it supports the undercurrents of the data engineering lifecycle.

Data management

💡 When evaluating a product make sure to ask the company about it's data management practices. This will allow you to know if a product is adopting best practices data management concerns.

- How are you protecting against breaches, both from outside and within?
- What is your product's compliance with GDPR, CCPA and other data privacy regulations?
- Do you allow me to host my data to comply with these regulations?
- How do you ensure data quality and that I'm viewing the correct data in your solution?

DataOps

💡 When evaluating a new technology, how much control do you have over deploying new code, how will you be alerted if there's a problem, and how will you respond when there's a problem? If the technology is OSS how will you handle issues, incident responses, and monitoring?

Data Architecture

💡 The main goals in accessing tools within the context of data architecture are avoiding unnecessary lock-in, ensure interoperability across the data stack and produce high ROI.

Airflow case study

💡 The airflow open source project is extremely active, with a high rate of commits and a quick response time for bugs and security issues, and the project recently released Airflow 2. Second, airflow enjoys massive mindshare. It has a vibrant and active community on many communication platforms. Third it is commercially available as a managed service. However, it has design downsides. It relies on nonscalable components (the schedular and backend database) that can become bottlenecks for performance, scale and reliability; the scalable parts of airflow still follow a distributed monolith pattern. It also lacks support for many data-native constructs, such as schema management, lineage and cataloging, and it is challenging to develop and test airflow workflows.

Software Engineering

💡 As a date engineer you should strive for simplification and abstraction across the data stack. Buy or use prebuilt open source solutions. Eliminate undifferentiated heavy lifting should be your big goal. Focus your resources - custom coding and tooling - on areas that give you a solid competitive advantage.

Conclusion

💡 Choosing technologies is a balance of use case, cost, build versus buy and modularization. Always approach technology the same way as architecture: assess trade-offs and aim for reversible decisions.