Architecting Data and ML Platforms - Notes

Preface

The book is about why a Data/ML platform can be valuable to a company, and all the different ways to build/evolve/utilize it.

It seems to recommend going to cloud hosting.

There doesn't seem to be much focus on GenAI, but there's mention of it.

Chapter 1

It focuses on platforms used by all parts of the company (finance, marketing, etc)

5 key parts of the data pipeline:

- collect
- Store
- Process/Transform
- Analyze/visualize
- Activate

And those can be thought of like a water treatment system.

Collect:

- 3 Vs of big data:
    - volume
    - Velocity
    - Variety (images, text, tables, flat files)

Store

- Scalability
    - Vertical
    - Horizontal
- Performance vs cost
- HA
- Durability
- Openness

Process/transform

If possible use simple SQL engines because SQL skills are cheaper/abundant. But if you need more advanced transform functionality or maybe even robustness (proper unit/integ testing), higher level language data engines for transforming like python, scala, java, can work.

Analyze/visualize
Many types of users, eg:

- business people, want visuals and roll ups
- Power users, want SQL like access
- Data scientists, will use ML techniques

Activate
This is where end users take action or make decisions based on the data.

3 categories of actions that can be taken:

- automatic (show X ad to user)
- SaaS integration(send list of prospective buyers to marketing dept.)
- Alerting (page pricing team, too many buyers)

Traditional approaches

siloed data, makes it hard to get insights from the breadth of data in a company, hard to do DR, scalability, business continuity, governance

There's a bunch of antipattern solutions for this in the next section. It basically is saying that ETL and data marts and data lake tech started the process of removing siloes, but there's problems added by them (which a unified platform presumably will solve in later chapters).

- ETL is an anti pattern! Solves by creating jobs for each of the data sources, all pushing relevant data to a single analytics data store, which is used by dashboards, users, etc. But ETL results in a **performance->cost->durability** tradeoff one must make, Further, there's more problems with it:
    - Data quality (extract often written by data consumers, not owners)
    - Latency (solved with streaming)
    - Bottleneck becomes data engineers who write code for ETL, and as data diversity increases, we need more of these teams
    - Maintenance
    - Change management
    - Data gaps
    - Governance
    - Efficiency/env impact
- Centralization of control is an anti-pattern!
    - The idea is to give 1 team the mandate to create a monolithic data platform that all the existing data must fall under
    - IT: too many technologies, unscalable Eng effort with new data sources/targets, governance is undefinable with so many data sources

    - Analytics: because of governance, can't give access to all data to 1 team, more ETL jobs needed to move data from disparate sources to targets
    - Business: can't trust data from many of the sources, hard to tell which ones, so hard to make business decisions based on untrusted data
- Data marts and Hadoop are an antipattern!
    - Data marts meant creating jobs to pull and process data from different places and put it into one place - a data lake
    - The engineering skill to do this is high, and there's not enough of it
    - Data lakes can be uncovered messes that few can understand

Further drawbacks of data marts / lakes:

- Business users know what'd be helpful, but don't have skills or access to data engineers who build/maintain the lakes
- Data engineers tend to focus on the data itself and are held at arm's-length from rest of or (business users)
- Governances means different things to different parts of organization that push/pull data from the lake - because of this, siloes actually get created again on top of data lakes (which were supposed to be part of the solution)

Data Lakehouse

Best features of lakes/warehouses - business users can use SQL queries, data engineers can write complex jobs to mix data. Also governance is better (somehow??).

But it's a technical compromise - limited to which data formats are used to store data, so optimizations that DWHs do aren't possible. Also the support for the types of SQL can be limited (eg geospatial, ML).

Industry is moving to this.

Data Mesh

Unifies data access while retaining ownership of the data in distributed domains.

Each team exposes a data schema for others to use to consume the data. Think of the data as 'products'. Each team who knows the data the best (they own the source) is also responsible for generating the value from the data (the schema, where people will use it). So where there was previously a challenge to trace a 'source of truth' for the valuable data (which is surfaced after various joins and transforms to business users for example), now the authoritative data source IS the product.

Hybrid Cloud

This section talks about

1. Why it's necessary
2. Downsides
3. How cloud providers make it possible (since it allows some companies/use-cases that'd otherwise stick with on-prem)
4. Edge computing, another incarnation of 'hybrid'

Why it's necessary:
Think of on-prem as just another cloud…then each cloud might have different advantages over others. On-prem might be necessary for strict geography/security/compliance for data locality reasons - e.g. finance and healthcare may never fully move to cloud for those reasons. Also one cloud might have tech that another just doesn't and vice versa, yet both are necessary/worth-it.

Based on that, here's why some companies can't move to cloud

- Burst capacity
- Data residency regulations
- Legacy investments
- Transition
- Best of breed

Here's challenges of having a hybrid cloud:

- Governance (different features in each cloud)
- Access control (on-prem kerberos/LDAP, public cloud IAM)
- Workload interoperability (some tech works here but not there)
- Data movement (expensive
- Skill sets (sometimes skills from one cloud don't translate to another)
- Economics (egress costs)

Things public clouds do to make it possible for companies to use hybrid (give us at least some of your workload):

- Choice - e.g. kubernetes is OSS, made by google - it actually gives a company choice if it's running in all the clouds, which is often the case. So then if company implements k8s on their on-prem, then it's easier to adopt GCP for some stuff
- Flexibility - at the framework level (by being locked in to cloud), or at the cloud level (by being locked into a framework like DB or Snowflake)
- Openness - e.g. Redshift queries are SQL, and Redshift has importers/exproters for data so it's easy to switch

It also talks about pros of using edge computing:

- Latency
- Less need for connectivity
- security/compliance

- Cost-effective solutionsj
- Interoperability

AI & Data Platform

Enabling AI pretty much requires that organizations adopt cloud. Cloud offers GPU availability without upfront crazy costs - even if you wanted to pay there's often no availability. Cloud offers scalability (compute, storage), easier operationalization, democratization of some AI capabilities (that means fewer skills needed while still being able to benefit from AI tech -e.g. An API to get audio translated - don't need to know AI to benefit).

The end of the chapter recommends adopting these key principles when building a Data Platform - basically the authors think these are the key drivers you should always come back to (first principles) if you're gonna build a data platform (prioritizing can slightly change for each org):

1. **Serverless access to data**
2. **End-to-end ML - data ingestion, storage, processing, training, evaluation, deployment, ops**
3. **Comprehensiveness - data storage, preparation, processing, sharing, BI and ML**
4. **Openness (enable OSS)**
5. **Growth (build for whatever scalability will be necessary for your org)**

Summary

Overall the chapter makes the case for why an organization should build (or adopt the idea of) a data platform, and how to think of what a data platform is and must be.

The why boils down to enabling all people in the organization to build/analyze/'activate' using all data in the organization that they should have access to (governance), and to do it in the most cost effective and bottleneck-reducing way possible.

The 'what a data platform is' boils down to 'a comprehensive platform that allows all departments of an organization to interact with data how they want to quickly, simply and reliably, and that reduces costs and engineering and/or decision-making bottlenecks to the maximum extent possible'.

The chapter also takes us through the path the industry went on to get to 'data platform'. Basically we started with on-prem siloed data that was brittle and where governance meant different things to different teams (sourcing or consuming the data) and where changes were bottlenecked at various places (engineering, those who know what changes are needed can't

talk to those who know how to make them).....to a unified data platform where governance and ownership from data source to ‘data product’ is achieved yet users have access to any data they want and should have access to in the way they want to access it (write code, or click dashboards and all in between). The end result is basically ‘lakehouse’.

The ML section was after that.

Self-Quiz

https://chatgpt.com/c/682ca608-1d9c-8004-bc52-5fc85273666d

Key takeaways:

- **Governance metadata must start at Collect.** Authors harp on *immutability* + *lineage* (think Kafka headers or Delta Lake `operationMetrics`).

Chapter 2

This chapter will go through a 7 layer pyramid of steps to go through to have a successful data platform project.

Step 1: Strategy and Planning

Goals, **stakeholders** and **change management processes**.

It talks through how important it is to be clear about what the goal of the project is - I love that it says usually it’s best to have an IT project focussed on a business goal, not something focussed on the existing system - it helps to stay ‘clear-eyed’ about what we’re trying to do. ‘Make queries faster’ or ‘make things more maintainable’ are not good goals - they’re IT-focussed, not business focussed. ‘Ensure queries complete within 10s at 100GB throughput and 1k QPS so that new user retention increases by 10% above baseline’ is much clearer and has a clear tie to a business goal. Projects tied to a business goal always have a much higher chance of success according to authors.

It’s important to find and talk to the right stakeholders - most likely from business or different business units - so that we minimize the chance of choosing a sub-optimal approach/strategy.

Even with the best clearest set of goals and the best set of stakeholders, a project can fail if there’s no plan of how to communicate that plan/execution to the rest of the organization - things

like training, asking how projects are assessed. A project will have more internal adoption with a good change management process/plan.

Step 2: Reduce TCO by moving to the Cloud

Benefits of doing so: operational cost (maintenance, engineers), right-sizing, autoscaling, serverless

Pretty straightforward - this section motivates moving to cloud and away from on-prem, similar to some sections of Ch1. One interesting insight is that they say you can often save 95% of TCO if you go straight from on-prem to serverless. It emphasizes savings from personnel and even from eng fungibility since cloud skills are common. Also, it says if you can't move all at once, or all at all, move the spiky/ephemeral workloads first.

Step 3: Break Down Data Silos

Basic steps: unify access (auth - use Okta and use cloud IAM since you should be on the cloud), choose storage tech (all data can be on same tech, like S3, but have separate governance owned by each data source team), create a semantic layer for users to actually access the data and apply governance to that layer.

Also, it says to bring compute to the data - kind of like how users would run DBSQL queries on Databricks on their data in S3.

Also says not to make copies of the data because it increases security and compliance (government data laws for example) risks.

Step 4: Make Decisions in Context Faster

Move from batch to streaming - to do it cheaply, treat streaming as a special case of batch, or vice versa. Use open source tech like Flink or Spark Structured Streaming - but use cloud services like AWS Kinesis to leverage it.

Step 5: Leapfrog with Packaged AI Solutions

This section basically lists out the different types of ML (predictive analytics, unstructured data understanding and generation and personalization) and products/services that serve the different layers of complexity for each (use an off-the shelf foundational model for zero-shot queries, or fine-tune it with your own data).

Step 6: Operationalize AI-Driven Workflows

This section talks about how to actually get AI solutions to actually be used. A few key topics:

- The right balance of automation vs. assistance - differs based on organization, sometimes this is a factor in competition between businesses in the same industry
- Building a data culture - creating the platform isn't enough, users need to know how to use it and be sold on using it and incentivized to use it
- Having personnel with the necessary skills to get all the steps done, including MLOps after a solution is deployed

Step 7: Product Management for Data

Maybe my favorite section - talks at length about the 'why' of building data products. 4 PM principles:

- Product strategy - how does dataflow today (before project), **what are key metrics, agree on priorities with stakeholders** and have a product roadmap
- Customer centered - **KYC and build for their skills & need**, don't make users do the 'change management' part
- Product Discovery - **interview users to find out what they need solved, whiteboard/prototype extensively before committing to a roadmap**
- Market fit - build only what'll be used immediately, standardize common KPIs, make it self-service

Summary

Key takeaways are to apply PM principles:

1. Have a product strategy
2. Be customer centric
3. Whiteboard and prototype to find solutions
4. Find the right balance between standardization and flexibility

Chapter 3

This chapter is about leveraging the existing team you have and the strengths they have to build a data team that works for your business.

First point is that data tool development is making so that each particular role in a data org can do more than they used to - data analysts will dip into data engineering, data engineers will dip into analytics and data science, data scientists will dip into engineering and analytics.

Great metaphor is a sports team that's great at defense - leverage teh strength, don't go trying to be the best offensive team when there are other teams with much stronger offense.

The chapter classifies orgs broadly into 3 categories:

- Data analysis driven
- Data engineering driven
- Data science driven

Data Analysis-Driven Organization

Characteristics: a mature industry, EDW and batch ETL, business intelligence (analysts comfortable with SQL and BI tools)

In the old world, data pipelines were expensive to operate and build, so the 'ET' part of 'ETL' was owned by a separate team, and when new data came in or new transforms were needed, they needed to be asked to make the changes, which often took time. Then the 'L' part would be available to data analysts to create value.

In the new world, we move to ELT, where all the data is funneled in, and then analysts can use common and user-friendly tools like SQL, dbt, etc. to find and query the data they need - no more asking swamped eng teams to build another transform job and then waiting for them to do it.

3 persona in DA-Driven org: Data analysts, Business Analysts, Data Engineers

Data Analyst
Receive, understand, fulfill requests from the business to make sense of relevant data.

This role needs to expand in 2 directions according to authors - DAs must learn more about the business (learn some BA skills) and DAs must be able to analyze and depict larger volumes of data (learn some DE skills).

Business Analyst
Domain experts who use data to act on analytical insights.

These days, this role can focus more on deeper understanding of the business and insights rather than admin/technical problems.

Data Engineer
Focus on downstream data pipeline and first phases of data transformation. Also manage governance and quality.

The authors urge us towards ELT and away from ETL where possible for DA-Driven orgs. The reason is because it gives more power/flexibility to the DAs to interact with the data at lower

latency: load the data sources in raw form into a data lake and also in a streaming system, and then allow DAs to query it using SQL in whatever way they want.

Principles for doing this: SQL as a standard, move towards a structured data lake and away from EDW, and schema-on-read first approach (as opposed to schema on write, or ETL).

Data Engineer-Driven Organization

This type of org is focussed on data integration - building integration pipelines. When data transformation needs are complex, DEs are needed to determine a successful data strategy to ensure we can build reliable systems cost-effectively.

Complex data pipelines can be expensive but gives you:

- Enriched data
- Training datasets of high quality
- Unstructured data that can go to ML models
- Productionization
- Real-time analytics (streaming)

DEs must know a LOT of things:

- DW and data lake solutions
- DB systems
- ETL tools
- Programming languages
- DS and algos
- Automating and scripting
- Container tech
- Orchestration platforms

DEs are responsible for ensuring that the data generated is needed by different business units and gets ingested where needed. They also help with deployment of artifacts produced by data consumers.

Authors have seen recently that some orgs will combine a squad of business units, DEs, DSs to own the responsibility of data sourcing through to decisions that make an impact on the business - similar to the ‘data mesh’ concept of ‘data as a product’ we saw in Ch1.

The authors give references architectures for all 3 cloud providers - the box shapes are almost exactly the same, it’s just each cloud has a different offering for each box component (e.g. BigQuery vs. Redshift vs. Synapse).

Data Science-Driven Organization

This type of org maximizes value of the data available to create a sustainable competitive advantage for the business. E.g. for a bank business, instead of assessing each business loan opportunity manually (a data analysis-driven org would do that), build a loan approval system that makes decisions on the majority of loans using some automated algorithm/model.

Key principles

- Adaptability - must be able to accommodate all types of users, even if they know nothing about ML
- Standardization - make it easy to communicate between teams & business functions and share code/artifacts
- Accountability - make sure there's clear responsibility for automated decisions that are made e.g. with audit trails and model explainability
- Business impact - because DS projects are expensive and tend to stop at 'pilot/PoC' stage, it's way more important to anticipate/measure business impact of new initiatives and focus on ROI
- Activation - the ability to operationalize models, scale to variety of users, facilitate seamless visibility/monitoring

4 main personas

- Data engineers
- ML engineers
- Data scientists
- Analysts

And the tech framework is basically a system to architect, train, test/evaluate, deploy, monitor and version/update ML models. Feature store, model registry, orchestration, and all the other usual components are required here.

Summary

Visions of the 3 org types:

- DA-driven: democratize access to data
- DE-driven: ensure data reliability & cost-effectiveness
- DS-driven: confer competitive advantage via business impact

Chapter 4

A Migration Framework

It's about migrating legacy systems to a new data platform.

Contents

- Conceptual model and framework for migrating
- Estimating cost of a migration
- Ensuring security & data governance during the migration
- Schema, data, pipeline migration
- Regional capacity, networking, data transfer constraints

Holistic view

- Business outcomes
- Stakeholders
- Technologies

Author says to modernize workflows first - this means thinking of it holistically, like 'what problem am i trying to solve? What does my user actually want to accomplish?' e.g. user wants to discover and target the highest value users of the product - what's the cheapest and simplest way to allow them to do that?

'Think of workflows not technologies' is an important aspect because by default, engineer will often think first of the immediate pain of the tools they're using and want to replace each of those with better ones - but this will only result in incremental changes, not transformational. To achieve transformational change, one must think of the entire workflow (and exactly which business purpose it serves).

High-level modernization points:

- Automated data ingest
- Streaming by default
- Automatic scaling
- Query rewrites, fused stages,etc.
- Evaluation (automated)
- Retraining (automated)
- Continuous training (automated)

Next point is to transform the workflow itself - 'if you can make the workflow something that's self-serve and ad-hoc, then you don't need to dedicate data eng resources to build it.

4 Step Plan

Prepare and discover

- interviews/questionnaire to stakeholders (across many depts, like IT, business, finance, compliance, security, etc.)
    - Use cases/workloads & their priorities - include compliance requirements, latency needs/sensitivities, etc.

- Explain expected benefits (e.g. query perf, data scaling, streaming)
- Ask them to suggest solutions already on the market
- Perform a TCO analysis (is this always a reasonable ask?? Maybe not…)
- Identify needs around training / recruiting (is this always a reasonable ask?? Maybe I do this part myself…)

Assess and Plan

1. Assessment of current state (isnpect the systems yourself - server configs, logs, job activity, data flow mapping, volumetrics, queries, clusters)
2. Workload categorization
    a. Retire
    b. Retain
    c. Rehost (aka “lift and shift” to cloud)
    d. Replatform (aka “move and improve” - partially change workload to improve perf or reduce cost and then move it to IaaS)
    e. Refactor (move to a PaaS solution like BigQuery/Redshift/Synapse)
    f. Replace - replace the whole thing with something off the shelf
    g. Rebuild
3. Workload clusterization
    a. Group 1: high business value, low effort to migrate (P0)
    b. Group 2: high business value, high effort to migrate (P1)
    c. Group 3: low business value, low effort to migrate (P2)
    d. Group 4: low business value, high effort to migrate (P3)

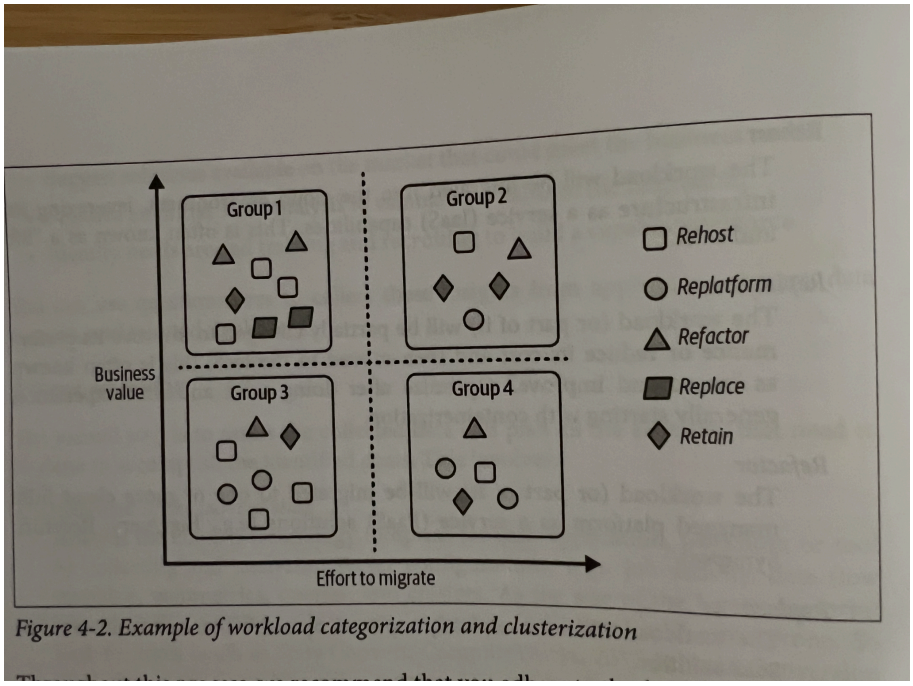

Figure 4-2. Example of workload categorization and clusterization

Recommended practices:

- Make the process measurable

    - Make sure stakeholders are sold, and have business KPIs they can evaluate results on
- Start with MVP or PoC
    - Be on lookout for quick wins (P0 workloads)
- Estimate effort required
    - Make a holistic project plan to define time, cost, people needed
- Over-communicate at milestones
    - Delivery value and instill confidence along the way with complete projects that people can actually use

Execute
At this point we should know:

- What will be migrated (whole workload(s) or part(s))
- Where they'll migrate to (IaaS, PaaS, SaaS)
- How to migrate (rehost, replatform, etc.)
- How to measure success
- How long it'll take
- What milestones we'll communicate at

Next, need to:

- Setup a 'landing zone' (target env where all workloads will reside)
    - Define target
    - Setup identity management
    - Configure authorization & auditing
    - Define & setup network topology
- Migrate to it
    - Multiple phases - small at first, then bigger, so we can gain experience & confidence
- Validate the migrated tasks/workloads

Optimize
Don't focus on perf of every individual component - instead, focus on system as a whole and identify potential new use cases to introduce to make it even more flexible/powerful.

Estimating Overall Cost of the Solution

Everything starts with an evaluation of the existing environment, as always.

Author says to come up with a CMDB - configuration management database. It should tell us all the hardware, software and other infrastructure components used in the system and how much

they cost. IT can come up with it manually, or there are tools to help automate or you can hire a 3rd party to audit.

Then author says it's often best to hire an external consulting company since they're more experienced at this? (Ok now this sounds like a sales pitch).

First, send out one of these to vendors:

- Request for information (RFI)
- Request for proposals (RFP)
- Request for Quotation (RFQ)

If the problem is difficult to describe or ambiguous, vendors often ask to to a mock-up. You can:

- Do a PoC - shows the key functionalities/changes/designs necessary to do the job, makes it clearer what the quote should be
- Do an MVP - ends with an actual usable product with the most important functionality, productionizable
- Both of ^^^ one after the other

Setting Up Security/Governance

Ownership and control of data may move to business units, but it's important that security and governance remain centralized - there needs to be consistency in the way roles are defined, data is secured and activities are logged. Without that consistency, it's hard to remain compliant with regulations.

3 risk factors that security/governance seek to address

1. Unauthorized data access
2. Regulatory compliance
3. Visibility

To solve for these factors, need a framework with these capabilities:

- Data lineage
- Data classification
- Data catalog
- Data quality management
- Access management
- Auditing
- Data protection

Agreeing with the intro blurb for this section, it is important to have an operating model and establish a 'council' responsible for ensuring various business teams adhere to the operating model.

To provide those capabilities, we need these artifacts:

- Enterprise dictionary - repo of info types used by the org
- Data classes - groupings of different info types
- Policy book - policies pertaining to each data class
- Use case policies - more nuanced policies that take into account more context (e.g. customer service rep currently working on request by customer X, so can see their address)
- Data catalog - manages metadata, lineage, quality, etc. for efficient search/discovery

A different axis to turn to is **data lifecycle stages (governance over the life of data):**

1. Data creation
2. Data processing
3. Data storage
4. Data catalog
5. Data archive
6. Data destruction

We must create policies for EACH of the above stages.

Also useful taxonomy is the different 'hats' individuals in the org may wear at different times:

- Legal
- Data steward (owner of the data, sets policies for specific pieces of data)
- Data governor - sets policies for data classes, categorizes a piece of data into them
- Privacy tsar
- Data user - maybe DA or DS or MLE who's using the data to make business decisions

Schema, Pipeline and Data Migration

Schema Migration

Best practice to migrate is to essentially use facade pattern. First migrate the model as-is to the target system (upstream and downstream both migrated as-is), then leverage the processing engine of the target environment to perform all the changes.

Pipeline Migration

There are 2 different strategies when migrating:

- You are offloading the workload
- You are fully migrating the workload

4 different data pipeline patterns:

- ETL
- ELT
- EL (data already prepared, no transform needed)
- CDC

The workload categorization earlier in the chapter also applies to pipeline migration:

4. Retire
5. Retain
6. Rehost (aka “lift and shift” to cloud)
7. Replatform (aka “move and improve” - partially change workload to improve perf or reduce cost and then move it to IaaS)
8. Refactor (move to a PaaS solution like BigQuery/Redshift/Synapse)
9. Replace - replace the whole thing with something off the shelf
10. Rebuild

Data Migration

Now that the new schema and pipelines are ready, we can migrate data. Main concern is data transfer - how to do it, costs, etc.

Planning - ask all the questions about the data being transferred up front, e.g. what are the datasets that need to move, what’s the security requirements of them, how big are they, where are they and where are they going, is it one-off transfer or ongoing, what’s the budget, what’s the time needed, etc.

People needed to be involved:

- Technical owners of the data
- Approvers
- Delivery / execution team

Regional capacity and network in the cloud

Find out and tell the hyperscaler / cloud management org what you need up front:

- Region
- Volume of the data
- Amount of compute needed
- Interactions it’ll have with other systems

3 main ways to get connected to the cloud:

1. Public internet (flaky, expensive, spiky)
2. Partner interconnect (more consistent and higher throughput, making deals can make it cheaper)
3. Direct interconnect (direct, physically connected to cloud provider, most consistent & can be cheapest)
    a. Only possible if both parties have routers in the same physical location

Transfer options

To choose from the different transfer options, consider:

- Cost
    - Networking
    - Cloud provider

    - Products
    - People
- Time
- Offline vs. online
- Available tools
    - CLI
    - REST APIs
    - Physical solutions
    - 3p off-the-shelf

Migrating stages
6 stages of migration:

1. Upstream code/arch modified to allow feeding target system
2. Downstream code/arch modified to allow pulling from target system
3. Historical data migrated, upstream process enabled to actually feed target system
4. Downstream system enabled to pull from target system
5. CDC to keep remaining legacy data in sync until it's retired
6. Downstream processes fully utilize target system

Final checks:

- Functional test of the migration
- Performance test of the migration
- Data integrity checks
    - Enable versioning
    - Validate the data

Important note from ChatGPT:

- Change-management & comms cadence – A migration dies not from tech faults but from stakeholder silence.

Chapter 5

Architecting a Data Lake

Why Data Lake
Data lakes today are meant for storing raw, ungoverned data from across the organization - they support Apache ecosystem tools. Data lakes initially gained adoption in order for organizations to store unstructured data - it was just storage. But then business units wanted to start using the data to make better decisions, faster - data lakes enabled advanced analytics and ML capabilities. However, on-prem ones are too high TCO, hard to scale, hard to govern / stay in compliance and are hard to change quickly (for agility). So cloud data lakes solve all those problems.

Design and Implementation

Design / Implementation depend on 4 choices:

1. Streaming or not
2. How will we do data governance
3. What hadoop capabilities we'll use
4. Which hyperscaler

Batch & Stream

4 storage areas in a data lake:

1. raw/landing/bronze
2. staging/dev/silver
3. production/gold
4. sensitive (optional)

2 architectures to allow both batch AND streaming:

- Lambda
    - New data comes into both batch layer and speed layer
    - Batch layer stores data in a dataset and then creates 'batch views' for serving to queries
    - Speed layer stores data in-memory and servers queries via a real-time view
- Kappa
    - New data comes into only a speed layer
    - Speed layer contains event storage which feeds real-time and batch views to queries
        - Data stored in an event streaming platform
        - If needed, data can be stored in a persistent dataset too

Data Catalog

A repository of all the metadata describing the datasets of the organization - which may be scattered across different axes:

- Which storage service: many sites, databases ,DWHs, filesystems and blob storage
- Which storage 'area': bronze, silver, gold, sensitive

Purpose of data catalog is to help rationalize various datasets across the org (find stuff), and even help deduplicate storage/compute.

A data catalog can also include data contracts, usually stored as JSON/YAML files.

Data Lake Reference Architectures

AWS Data Lake Architecture

- Data source: S3, Relational DB, NoSQL DB
- Storage zone: S3

- Data catalog: Lake Formation
- Analytical services: Athena, Redshift, EMR

Azure Data Lake Architecture
- Data source: Azure Data Lake Storage Gen2
- Data transformation/ingestion: Azure Data Factory
- Data catalog: Azure Purview
- Analytical services: Azure Synapse Analytics, Azure Databricks
- Data Visualization: Power BI

GCP Data Lake Architecture
- Data source: Cloud Storage
- Data transformation/ingestion: Data Fusion, Dataproc
- Transform/real-time-to-hadoop integration: pub/sub
- Data Catalog: Dataplex
- Analytical services: Bigtable(real-time), bigquery(analytical)
- Data Visualization: Looker/Looker Studio
- Data workflows: Composer

Integrating the Data Lake

The superpower of a data lake is the ability to connect data with an unlimited amount of compute.

APIs to extend the Data Lake are very important - everything from HTTPS, gRPC, standard ETL to pytorch and tensorflow.

ACID and frequent data updates
To be able to have ACID guarantees and manage frequent updates, there's a layer on top of a Data Lake that can be useful, and it's provided by 3 options:
- Apache Iceberg
- Apache Hudi
- Delta Lake

Some benefits:
- ACID
- Any file size (HDFS can't do that well…?)
- Logging/audit of changes
- Parquet storage
- Governance at row/column level

Additional benefits:
- Partition evolution (evolve how a single file gets split)
- Schema evolution

Jupyter Notebooks
The best Swiss Army knife an organization may leverage for data analysis.

They've become the de facto standard solution for interactive data analysis, testing and experimentation.

Managed versions on clouds: AWS Sagemaker, Azure ML Studio, Google Cloud Vertex AI Workbench.

Democratizing Data Processing and Reporting

Because the value of data comes from enabling decision makers and users to make informed decisions, it makes sense to make it accessible and usable by any authorized individual. That essentially means switching from an IT-driven approach to a more democratic approach when building a data lake.

In the past, there'd be a 'trust' bottleneck where 1 person/team can convince business users of a given data product/report/output that it's correct - they'd be able to answer questions like 'where did the data come from?', etc. When it's handed off to an IT-team, they might not be able to answer those questions. Nowadays people are generally more digitally experienced and that helps a lot with shifting this - we make final users able to dig into their data in an autonomous way.

That means data processing, cleansing, wrangling, which used to be done by experts (data engineers for example), can now be done directly by final users. This works because the underlying data (often in s3 or s3-similar services) can be accessed by many different tools that are available now: Talend Data Preparation, Trifacta Dataprep, scikit-learn, Tensorflow, PyTorch, Hive, PrestoSQL. These can serve SMEs, data scientists and business analysts respectively.

Data Stewardship: process of managing and overseeing an org's data assets to ensure that business users have access to high-quality, consistent, easily accessible data. 3 factors:

1. Identify key stakeholders
2. Defend the data (exfiltration, other attacks)
3. Cooperate with others to unlock the value of the data

Data Ingestion
Data lakes can become swamps of unused data if not managed properly because nowadays anyone can add data into it, not just data engineers.

Best practices to leverage during ingestion:

- File format
    - For readability use: CSV, JSON, XML
    - For performance:

        - Avro RPC, low latency use cases
        - Parquet and ORC (columnar) for query use cases
- File size
    - For hadoop world, larger file size is better
    - For small files, try to consolidate
- Data compression
    - Need compression/decompression on the fly
    - Don't use zip
    - Snappy is one option
- Directory structure
    - Depends on use case

Many different ways to ingest, e.g.:

- Scripts
- Directly using APIs/connectors
- Stream it into the platform
- Raw files into HDFS

ML in the Data Lake

You can store any type of data in its raw format in a data lake - e.g. unstructured images, videos, natural language, etc. in their typical formats (JPEG, MPEG, PDF, etc.). Perfect for using it to train ML models.

The ML framework to be used depends on the format. For structured data, Spark, XGBoost, LightGBM - they can directly read/process CSV files. For unstructured data, TensorFlow/PyTorch - they can read most image and video formats in native form.

There's an efficiency tradeoff though - directly reading JPEG/MPEG leads to ML training being IO bound. So the data is often extracted and stored as TFRecords or similar, which help optimize GPU utilization. GPU manufacturers also provide capabilities to read common formats (e.g. Apache Arrow) directly from cloud storage into GPU memory.

Inference directly on raw data also is straightforward because the ML frameworks read raw data formats.

Chapter 6

Innovating with an Enterprise Data Warehouse

Which to Choose - Data Lake or DWH

Comes down to whether the organization is engineering/data-science first (choose a data lake) or analytics-first (choose a DWH).

A Modern Data Platform

When you start a large tech project, ask first:

- What business goals are you trying to accomplish
- What are your current technologies challenges
- What technology trends do you want to leverage

Common Organizational Goals

- No silos
- Democratization
- Discoverability
- Stewardship
- Single source
- Security and compliance
- Ease of use
- Make DS teams more productive (they're expensive)
- Agility (allow users to make decisions faster)

Common Tech Challenges

Overarching themes (read about them in previous chapters): **volume, variety, velocity.**

- Size and scale
- Complex data and use cases
- Integration
- Real time

Common Tech Trends/Tools

- Separation of compute and storage
- Multitenancy
- Separation of authN and authZ
- Consistent admin interface
- Multicloud platforms
- Converging data lakes and DWH
- Built-in ML
- Streaming ingest
- CDC

Hub and Spoke architecture

If you're building a data platform around a DWH, hub and spoke is the ideal architecture. The DWH is at the center, surrounded by APIs (query, Streaming, Load, Storage).

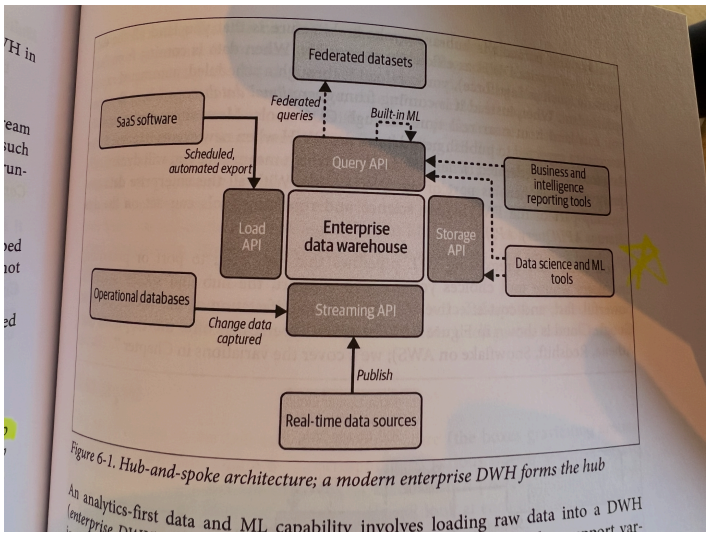

Figure 6-1. Hub-and-spoke architecture; a modern enterprise DWH forms the hub

An analytics-first data and ML capability involves loading raw data into a DWH

Key behind hub and spoke is you land all the data inthe DWH as efficiently as possible.

Main components: ingest, business intelligence, transformations, ML.

Data Ingest

3 ingest mechanisms:

1. Prebuilt connectors
    a. 1st party ones, like where cloud makes it easy to ingest, e.g. BigQuery has a connector for salesforce, Google Marketing Platform and Marketo.
    b. Also can use third party vendors like fivetran, which make connectors for various sources to various targets
2. Real-time data
3. Federated data
    a. Query the data as if it's in the DWH but it's stored elsewhere - usually needs to be common format like Avro, CSV, parquet, jsonl. With this one the compute for query is done on the DWH
    b. Similar is 'external query'. With this one, the compute is run on the external data source

Business Intelligence

Data analysts need to rapidly be able to derive insights from the data, so the tools need to be self-service, support ad-hoc analysis and provide a degree of trust.

Several capabilities it must provide:

- SQL Analytics
    - Always best to have the BI tools push all queries to the DWH, rather than something else on top (OLAP cubes)
- Embedded analytics
    - To serve a larger-scale number of users who want to see their specific use case of data - have them use embedded analytics that are already built-in to the tools they're using ,as opposed to something more flexible like a grafana dashboarding thing that's hard to support for so many use cases. Think simplicity for these users
- Semantic layer
    - A layer that translates the nomenclature of the table creator to/from names adopted by data users

Transformations

Sometimes it's useful to not have user need to transform the data to what they need at the time they need it - for this, use transformations - there's different types with different pros/cons:

- ELT with views (virtual table sort of, and query and processing into that 'shape' of that virtual table happens at query time)
- Scheduled queries (if tables updated infrequently, extract data into downstream tables periodically)
- Materialized views (upon first read query, store that data for any subsequent reads of the same view)

Security and lineage

Need to keep track of (data lineage) :

- Who is accessing what
- The origin of the data
- The different transformations
- The current physical location

Data quality also is important - has a few common techniques to support it:

- Standardization
- Deduplication
- Matching
- Profiling and monitoring

Organizational Structure

Basically, engineering owns:

- Landing the raw data from a variety of sources into the DWH

- Ensuring data governance
- Workloads that cross business units or require specialized engineering skills
- Common artifact repos (data catalog, source code repo, secret store, feature store, model registry

And business unit(s) owns:

- Landing data from business-specific sources
- Transforming the raw data to usable format for downstream analysis
- Populating the governance catalogs and artifact registries with business-specific artifacts
- Reports, ad-hoc analysis and dashboards for business decision making

Usually it's best to get different parts of business / different business units to adopt the same underlying DWH tech - makes data sharing easier. Sometimes harder to do, but it can be really not worth it to support more than 1 system in terms of maintenance and alignment.

DWH to Enable Data Scientists

Data analysts support data-driven decision making by doing ad-hoc analysis and creating reports and then they operationalize the reports using BI. Data scientists aim to and scale data-driven decision making using statistics, ML and AI. They both need to interact with the DWH in various ways to query the data and get access to the data at a low level. The primary tool they use is **notebooks**.

Interfaces

- Query
    - This is necessary for while they're exploring / developing in the notebook - they need access to the data in an interactive way - basically, they need a fast way to run SQL queries
    - The combo of DWH backend and notebook for programming and visualizing as frontend is powerful (notebooks backed by the DWH).
- Storage API
    - This is necessary for when automating their work - speed/ throughput is paramount, so Query API doesn't do it
    - Need to be able to do parallel reads from multiple threads to stream while the accelerator trains for example
        - *S3 client is important!*

ML Without moving your data

You can do these things actually within some cloud data warehouses:

- Training simple ML models using SQL

- Serving ML models as part of a larger ML workflow that includes the DWH
    - Export trained ML models for deployment
    - Incorporate ML training within the DWH
    - Invoke ML models as external functions
    - Load ML models directly into the DWH to invoke

Chapter 7

Converging to a Lakehouse

Is it possible to make both data lake and DWH coexist to serve both types of users?
Yes - lakehouse architecture does this and there are 2 broad variants.

The Need for a Unique Architecture (Motivation)

User Personas

- BA - typically close to the business, focus on deriving insights from data. Traditionally have their data prepared via ETL tools based on DA requirements. Typically know SQL.
- DE/DS - typically closer to the raw data with tools/ability to mine it. Transform the data in various ways to make it accessible by business. Also experiment and use it to train ML models and for AI processing. They find answers for business, but also find useful questions for business to ask. Typically know programming languages.

Anti-patterns

- Disconnected systems - opportunity cost spending resources on ops rather than business insights. Causes data quality / consistency problems. Causes data puddles
- Duplicated data - leads to generation of data silos. Results in costly to maintain data marts, ETL jobs that need updating and business users needing engineering effort for any new type of insight. Data governance repeated in the lake and the DWH
    - Proliferation of data
    - Slowdown to TTM
    - Limitation in data size
    - Infra/ops cost

Converged Architecture

Converging DWH and data lake allows us to serve DS, DA, BA alike. The key factor is that when we converge, they both share common storage underlying. And one of the key things making convergence possible is: separation of compute and storage - allows cloud providers to bring different types of compute to common storage (also hard to do that on-prem, because

machines have to be procured months in advance). Then, 2 different engines (SQL & distributed programming) process data without moving it around.

Two Forms

Data Lake First
Data stored in open source format (parquet, avro) on cloud storage (s3). You can obviously use Spark to process it. At same time, Spark SQL provides interactive queryability. In addition, DWH tech can support running SQL directly on the datasets without copying/moving. This is what Databricks does.

DWH First
Data stored in highly optimized DWH (e.g. BigQuery format, Snowflake format). You can run SQL on it obviously. In addition, they support the use of compute engines like Spark directly on the native data.

Both are compromises - data lake first make SQL slower/costlier, DWH first makes Spark/ML slower/costlier.

How to Choose
Choose based on the skills of the majority/most-important users. Data lake first best for proficient programmers, DWH first is best for users who want to interact with data to gain insights, especially business insights.

Other Factors for choosing: volume of data ingested, whether streaming is needed, how much of data structured/semi-structured vs unstructured, whether it's for org in a regulated industry (finance, medical).

Lakehouse on Cloud Storage (Data Lake First)

Use cloud storage (s3) as underlying storage for both data lake and DWH access. High throughput is a benefit. Both streaming and batch data would have a historical ledger stored as raw tables for consumption by DS's making ML models. We can also leverage scheduled queries and event-based Lambda architecture for data ingestion.

Single analytical engine that can access both data lake and DWH data. This makes architecture more streamlined/efficient (rather than having a separate SQL engine).

Data is stored in one common place. Governance rules can be applied to all data across data lake and DWH views/APIs/UIs/engines.

Migration strategy to this architecture is similar to back in Chapter 4. Start with quick wins to demo benefits.

To **Future Proof** it, make sure real-time and consistent access to data is possible - streaming + ACID (Iceberg/Hudi/Delta Lake).

SQL-First Lakehouse (DWH First)

One advantage is that business users can carry out orchestration and ML - **it does a better job of democratizing access to data than the Data Lake First approach**, since there are many more users who can do SQL and it's easier to learn than programming languages.

Needs to provide standard SQL-ability, but also Spark env integration, ML capabilities and streaming. Data flows from original sources through both data lake and native DWH storage.

DWH storage split in 3 dimensions:

- Raw - data comin in as-is (batch OR streaming)
- Enriched - raw data with 1st layer of transformation
- Curated - enriched data ready for final transformation

Spark (ETL) or SQL (ELT) can do all transformations. SQL is preferred approach to transforming data. For more complex processing, DWH-first approach still needs to support structured programming language like Spark. Also important to support legacy data processing jobs still, so existing code can be leveraged.

Again, a key advantage of lakehouse paradigm is that it's forced vendors to develop many complex solutions very easy to use for a broad set of users - particularly true for using ML. There's now many ways to use ML within DWHs, even within SQL-like syntax.

Still, most ML tasks need to support Python frameworks since that's what DEs/DSs are comfortable with.

Again, key ingredient is to be able to leverage Spark.

MLOps also important with DWH first approach: data versioning, data lineage, model updates, etc. Many cloud solutions have this like Databricks MLFlow, Google Vertex AI, Amazon Sagemaker.

Migration like with DL-first approach, is an iterative process. First step is to enable ingestion to DWH (batch & streaming). Then, cut out external analytical engines and start elevating SQL engine of the DWH as the main one. Pay attention to how we'll migrate DL workloads to DWH - make sure there's a built-in programming language module like for Python or Spark connectors/engines, that can work directly on the native storage format. In the end, most end users/analytical teams will use the curated data, while most DE/DS will prefer access to the raw/data-lake and enriched data.

Benefits of Convergence

- Time to market
- Reduced risk (can continue leveraging existing tools without rewriting them)
- Predictive analytics - real-time decision making using fresh data
- Data sharing
- ACID transactions
- Multimodal data support
- Unified environment
- Schema and governance - as data changes, can have a holistic view of governance
- Streaming analytics - realtime analytics for use cases where need very low latency

Chapter 8

Architectures for Streaming

The industry trend is away from batch and towards streaming.
This chapter:

- Different streaming architectures, especially deep dives into 2 of them
    - Micro-batching
    - Streaming pipelines
- How to support real-time, ad-hoc querying in both of them
- Architecting a system to autonomously take actions when certain events happen

The Value of Streaming

Example: 2 businesses assess/approve loans - 1 takes 3 days, the other in minutes. The latter has a competitive advantage.

What's even better than faster decisions is being able to make decisions **in context**. Decisions while the event is proceeding is significantly more valuable than even a few minutes later (e.g. fraud prevention).

Industry Use Cases

- Healthcare - personalized IoT medical devices, real-time patient monitoring
- Financial services - prevent fraud
- Retail - personalized ads, real-time inventory
- media/entertainment - personalized content/ads, minimize churn

Streaming Use Cases
Think of streaming as data processing where the data has no bounds - unbounded datasets. Two challenges there:

1. data set infinite, never complete - all aggregates can only be defined within a time window
2. Data in motion, held in temp storage, so can't use conventional programming techniques like file handles to process it

4 categories of streaming use cases, in order of complexity and value:

1. Streaming ingest - store it
2. Real-time dashboards - view it
3. Stream analytics - transform and notify me about it
4. Continuous intelligence - automatically act on it

Streaming Ingest

Can be done in 2 ways:

1. *Streaming ETL* Aggregate events, periodically write them to persistent store
2. *Streaming ELT* ingest data directly into lake/warehouse and expect clients to transform

Streaming ETL

This works if your goal is to provide business with more timely data. Need to store the data somewhere that can be accessed by end users - so goal here would be to land it in a DWH. Wherever the data gets generated, have a local agent on the application (similar to how prometheus/monitoring 'pull' architecture works) that makes the data available for real-time monitoring/querying. Local agent can do 2 things:

1. *Micro-batching*: Write data periodically to a cloud storage file, and when a new file is written trigger a lambda function or something similar
2. *Streaming pipeline:* Write each event to a message queue and push those to a streaming pipeline that pushes them to final storage destination

To choose, consider latency requirement - DWH load jobs queue, so there's added latency with #1. But micro-batching is usually less expensive.

If you don't need to transform at all, consider *streaming ELT* instead.

Streaming ELT

Use this if you can't anticipate how users will want the data transformed - usually an org moves toward this as the use case volume/variety grows so you can't anticipate what they are and also start to require business-specific knowledge that business users are more suited to do.

Again there's a local agent, but it **loads** or **inserts** raw data into DWH. Big drawback for ELT is the data can be large increasing costs.

Counter-intuitively, the more data you have, the more cost-competitive streaming ELT becomes. This is because as data volume gets larger, the ‘transform’ part takes longer and hits a wall where it’s no longer meeting the requirement. Then you can scale up since it’s an embarrassingly parallel problem. But instead of scaling up periodically, use micro-batching - cost of compute is roughly the same, but you get reduced latency, increased frequency, the load is spread out and spikes are better handled. In addition, the org gets more timely and less stale transforms/reports/jobs which can have a huge business benefit for no extra cost.

Streaming Insert, Edge IoT

If cloud provider provides a streaming API, you can stream data to the DWH directly from the application (no ‘local agent’ or ‘sidecar’, just a client library being called in the app code).

For IoT, usually there’s a more specialized setup where an edge SDK is used to do local processing, data management and ML inference and then ‘insert’ data into the server-side ‘IoT Core’ which is a component sitting in front of the DWH. Usually SDKs will support standard protocols like MQTT - always use the standard ones so it’s easy to change your architecture.

Streaming Sinks

Usually you’ll want to land streaming data directly into the DWH, but there are 2 exceptions: **unstructured data and high-throughput streams.**

If streaming video from the edge, the camera will likely support a real-time streaming protocol like WebRTC. On the server side, something like Google Cloud Video Intelligence will convert from the live streaming protocol to a decodable video stream and write the stream to files on cloud storage.

DWHs are general-purpose answers for persistent storage and interactive, ad-hoc querying. But they’re not good for high throughput or low latency use cases. Instead use NoSQL databases, though they’ll cost more.

You can also stream files to cloud blob storage as long as you don’t need immediate querying capability - this would be the strategy if you have a data lake but not a lakehouse. Make sure to decide how to shard the files for your use case.

Dashboards & Live Querying

Dashboards showing stats/trends on the incoming data are often useful.

For live querying, it’s always preferable to query the DWH live instead of storing materialized views and such - it can get expensive if you do that too much.

Streaming Analytics

When implementing a dashboard you might also need to enable alerting capabilities that are triggered by automated insight/prediction extraction and that flow to users. This is stream analytics.

It's useful in these 3 situations: time-series analytics, clickstream analytics, anomaly detection.

Time-series Analytics

- Most common
- Validate data periodically or compute time-based averages
- Stream processor is capable of applying time windows to incoming event stream to compute stats like avg. speed or w/e

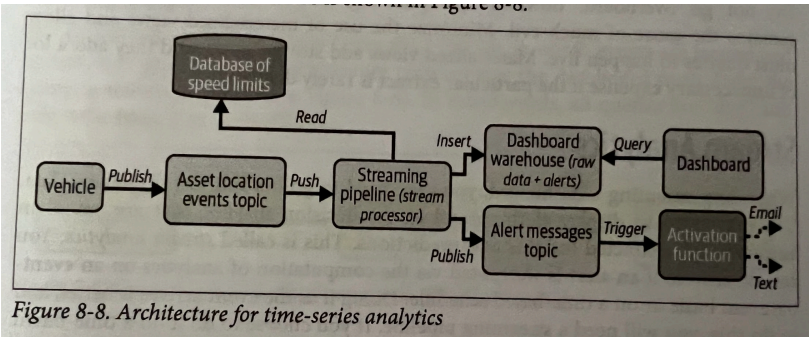

Figure 8-8. Architecture for time-series analytics

Clickstream Analytics

Clickstream consists of a sequence of events carried out by visitors within an app/website. To collect data, the website is instrumented. Something standard like Google Tag manager to instrument it, then use something like Google Analytics would provide a way to export clickstream data into BigQuery, and then you can transfer it to any other DWH. Once it's in the DWH you can analyze it with SQL.

Combined use of all the data availableacross all channels to tie together what is likely the same user is called **identity stitching.** The lakehouse where resulting set of unique user IDs and corresponding attributes is stored is called the **Customer Data Platform (CDP).**

It's common to build a postprocessing "backfill" pipeline to handle cases that are unique to your org.

Anomaly Detection

Involves identifying unusual patterns as the data is arriving.

Signature-based patterning is the primary technique but for attacks never seen before, use unsupervised learning - clustering - to detect events far from existing clusters and assume they're suspect and need further analysis or to be stopped.

Anomaly detection has 2 streaming pipelines:

- 1 to compute clusters
- 1 to compare incoming events to the recent clusters and raise an alert

Resilient Streaming

There's gonna be malformed data. In past with batch, it'd just stop the job, you'd make a fix, and then restart the job. In streaming, the pipeline must continue running.

So setup a **DLQ dead letter queue.**

When there's an issue detected, 2 options to fix and apply the fix to the pipeline:

1. Update - keep in-flight data running, update the pipeline logic, and it'll continue writing to the same persistent store. Only works if changed pipeline output is compatible with the persistent store. Ensure **exactly once** processing
2. Drain - the old pipeline stops pulling from its input sources, sends new events to the new pipeline. Only do this if new persistent storage is not compatible with the old one. Ensure **at-least once** processing

Continuous Intelligence Through ML

Not necessary to have a human in the loop to make decisions. As data volume grows, it's common to move to a system of **human over the loop** - a human supervisor can override an action/decision, but a system is designed to operate autonomously without any human intervention. This is **continuous intelligence.**

To do this:

- Train ML model on historical data (training)
- Invoke it as events come in (inference)
- Take actions based on predictions

Training

Train on 'recent' data. Whatever 'recent' means for the use case. Train on data you'll actually see in production.

Windowed training is to train on some size of a historical window of data and do it often.

Scheduled training is to train on the most recent x time units of data periodically but not too often.

It's important to have **continuous evaluation** so you can tell when you need to retrain (detect **data drift**). Also useful to be able to detect **feature drift**.

Streaming ML Inference

You can sometimes load the model into the prediction pipeline and run inference there - but only if the model is small enough. Otherwise you must deploy the model to an endpoint and invoke it as a microservice. But ML inference is often more efficient when done in batches - so pass in small batches.

Automated Actions
If you need the predictions to sometimes trigger an action, you could use serverless functions like Lambda.

Additional Notes (ChatGPT)

- **Event time vs. processing time** and **watermarks** (plus "allowed lateness") to handle out-of-order events.
- **Window types:** tumbling, sliding, session; plus early/late/on-time **triggers**.
- **Delivery semantics:** at-most/at-least/exactly-once; lean on idempotent or transactional sinks (e.g., upserts) for true end-to-end exactly-once.
- **Schema management:** Avro/Protobuf + **Schema Registry** with backward/forward compatibility so streaming jobs don't break.
- **Stream–table duality & CDC:** change-data-capture from OLTP (binlog) as a first-class streaming source; materialized views rather than overusing static aggregates.
- **Scalability:** partitions, backpressure, checkpointing, and hot-key mitigation (key redesign, sub-keying, state sharding).

Chapter 9

Extending a Data Platform Using Hybrid and Edge

Why Multicloud?

A single cloud is simpler and cost-effective because of **simplicity**, **learning curve** and **cost (simpler security & contracting, IT and legal, scale discounts).**

Multicloud is inevitable often because of **acquisitions**, **best/most-familiar tools** for employees and because you need to be **supporting customers** in different clouds sometimes.

Multicloud could be strategic - here are some possible reasons to adopt it:

- Fear of lock-in / Exit strategy / Leverage / Portability
- Commercial (marketing might use multi-cloud as a selling point)
- Regulatory requirements
- Employee knowledge

Multicloud Architectural Patterns

Single Pane of Glass

We want to develop a solution that enables data analysis across several data silos, maybe in different clouds and regions - to do this we need to be **cloud agnostic**.

2 approaches:

- BI tool-based approach - user uses a chosen BI tool and the tool does the work of connecting different data sources, retrieving data, executing queries remotely and aggregating results.
- Processing engine-based approach - users use BI tool, which uses a local process engine to connect with data sources, retrieve data, execute queries remotely and aggregate results.

Write Once, Run Anywhere

This approach gains cloud optionality.

Ways to do this:

- Managed OSS
- Multicloud products (snowflake confluent kafka, databricks, etc.)
- Multiple runners - use OSS to implement, but managed services from multiple clouds to run workloads
- OSS abstraction layer - e.g. use langchain as a consistent LLM prompting interface and have cloud LLM APIs behind it instead of using them directly
- OSS on IaaS - do it all yourself, just use commodity cloud compute instances (authors say to avoid this unless you have the scale to make it cost-effective)

Bursting from On Premises to Cloud

This is the easiest way to get started with cloud workloads. Prime candidates for bursting are large one-off jobs.

This approach uses Hadoops distributed copy tool, DistCp, to move data from on-prem HDFS to cloud blob storage. Then PaaS Spark jobs are run to process the data. Once it's done you can keep the blob buckets for future jobs or delete them.

Other uses cases:

- Test version upgrades
- Experimentation and testing new tech

Pass-Through from On Premises to Cloud

This pattern is complementary to the last one. Similarly, use DistCp to move data to cloud blob storage. Then use cloud process engines to process the data in ways you can't/didn't do on prem, maybe because those process engines don't exist on prem.

Some workloads that can be enabled this way:

- Process ORC, parquet, avro files and leverage **federated queries**
- Join on-prem datasets with data in the cloud
- Build models based on on-prem data leveraging AI/ML tools
- Run batch predictions at scale on the on prem data

There are existing tools like Informatica, Talend, TIBCO, MuleSoft that can facilitate integration of data sources and maintain synchronization.

Data Integration Through Streaming

Change Streaming is the movement of data changes as they happen from source to destination - It's basically CDC.

CDC is an approach to data integration that enables organizations to integrate and analyze data faster, using fewer system resources. Highly effective for limiting impact on the source data & system and eliminates need for bulk updating.

Most common cases, in order of commonality:

- Analytics
- DB replication and sync
- Event driven architectures

Adopting Multicloud

Framework

To translate multicloud strategy into a multicloud IT architecture, you can use TOGAF (The Open Group Architecture Framework) to:

- Identify business needs
- Define what data is needed to support the process
- Define how this data is processed by the applications

Time Scale

There are roughly 3 possible time scales to think of for multicloud adoption:

- Org may never fully migrate (regulated industries for example)

- Org may need multi-year journey to migrate - will have hybrid as end state for years
- Not currently ready to migrate, but can't meet business SLAs because of ad-hoc large batch jobs

Define a Target Multicloud Architecture

Here are the steps to adopting multicloud:

1. Define a strategy - identify the drivers for multicloud (e.g. cost, lack of features, openness, regulatory compliance, etc)
2. Develop the team
3. Assess the current situation - understand what is currently operational. Decide what's best, e.g. lift and shift to maintain existing solutions or replatform/redevelop to get most from cloud provider?
4. Design the architecture
5. Prepare a migration plan
6. Build the landing zone
7. Migrate and transform

Why Edge Computing?

Edge computing is the architectural pattern that promotes the execution of the data processing right where the data is generated. It helps organizations handle workloads where data must be processed in place (or with very low latency) or when activities need to handle short or long network partitions (when devices can't connect to cloud).

The section goes through an example where a manufacturing company wants to use ML to detect and stop defective items. The challenges:

- Bandwidth - system needs high res pictures
- Latency - predictions/outcomes need to happen within milliseconds to stop defective items
- Offline - happens in factories, which are often placed away from cities and where there's bad connectivity

Use Cases

- Automated optical inspection
- Improved security
- Agriculture
- Healthcare
- CDN (cache info at the edge to reduce latency when reading it)
- Immersive interactions (VR/AR)
- Smart cities
- Traffic management system

Benefits

- Reliability
- legal/compliance - GDPR, data location laws
- Security - reduced attack surface (reduced risk of DDoS, increased data protection)

Challenges

- Limitations in computing (CPU/GPU/TPU/etc.) and storage capabilities
- Device management / remote control
- Backup and restore

Edge Computing Architectural Patterns

It's important to have a clear strategy in mind when defining an edge computing architecture. There are, broadly, two types of edge computing architectures:

1. Where devices are smart
2. Where a smart gateway is added at the edge, to talk to 'dumb' devices

Either way, ML activation works similarly.

Smart Devices

This architecture is straightforward yet expensive. The devices can be called 'nodes' and their hardware can vary a lot.

Smart Gateway

'Dumb' devices/sensors connected (wired or wireless) to a smart gateway that can execute the logic on their behalf. Preferred approach when dealing with many sensors in the same location, like in a factory, or smart home.

This introduces security challenges because smart gateway is a SPOF.

ML Activation

Review different steps in the ML process:

1. Data collection (pictures of items)
2. Data analysis (check quality, add labels)
3. ML model development/training/testing
4. ML model deployment
5. Feedback data collection

ML modeling steps impose certain requirements on the edge architecture:

- Data collection requires the ability to store collected images for a long enough time in between connections to central cloud
- Data analysis can happen in the cloud
- ML model development can happen in the cloud, but testing requires a simulated edge environment
- Deployment of the ML model to the edge may require an upgrade of hardware components to include ML inference chips
- Feedback of data collection requires tracking of when predictions are detected as wrong/off

Adopting Edge Computing

Authors go through an example of a can making company that wants to implement ML at the edge to detect defects. They want to fix issues and correct inaccuracies in their manufacturing process.

They decided on a 3 step journey:

1. Improve overall system observability
    a. Have a local architecture (devices/sensors connected to a gateway)
    b. Have a centralized architecture too (gateway connected to cloud)
2. Develop automations
    a. The team used data to understand:
        i. How workers interacted with the machines
        ii. Correlation between configuration of the actuators and defects
        iii. Average inactivity time due to a mechanical problem
        iv. Ideal machine configurations when using specific raw materials
3. Optimize the maintenance
    a. Maintenances carried out in 2 ways:
        i. Planned
        ii. Unplanned
    b. They developed a predictive ML model algorithm to calculate the probability of a failure in a machine

After that successful project, the company decided to focus on reducing the cost of the quality checks by automating as much of the process as possible.

Chapter 10

AI Application Architecture

This chapter has high-level decisions you should make on architecture and frameworks when building an AI and ML app. It talks about what types of problems ML/AI is good at solving and what it's not. It also goes through the decision process for deciding to 'buy, adapt or build'. If you build, there are several architecture choices and the chapters a bit about that.

This chapter is about architecture decisions at the app level - the next chapter is about the platform on which MLEs and DSs build.

Is this an AI/ML Problem?

ML is a subfield of AI. Deep learning is a subfield of ML. Generative AI is a subfield of Deep learning. In GenAI, the output of the model is text, images, audio, video, etc.

At time of writing, the authors talk about how auto-regressive models work because that's what all/most GenAI models do.

Because LLMs can't remember all possible word sequences in context, they instead interpolate between probable words and contexts. Research is needed to determine whether models can truly generalize beyond the text they are trained on.

LLMs hallucinate - they may produce text that is syntactically correct but illogical or untrue. The amount of work needed to check for accuracy is something to be considered when planning an app in production.

To reduce the amount of nonsense generated, RLHF is used - training a model to choose between generated text docs based on human ratings.

Weaknesses of LLMs

- If the input sources are wrong ever, the output can be too
- Models are better in domains where there is a lot of text that is published in digital formats (coding, politics….not so great at anthropology)
- Hard to tokenize unusual numbers or names - not great at math and can be inaccurate with factual info like authors of articles
- Non-deterministic use cases only

Use cases (where LLMs are good)

- Domain-specific assistants
- Code snippets
- Doc summarization
- Content generation

But creating an AI app is very hard.

The most compelling use cases of ML happen when you have many of these conditions in operation:

- Repeated decision - frequent decision/'brain task' that gets expensive because of how often a human has to make it/do it. Conversely, not much business benefit to automating rare decisions
- Labeled data
- Fault-tolerant use case
- Logic that is hard to articulate - the more complex the logic is to articulate, the better ML is at it. When it's easy to capture logic, use traditional programming methods
- Unstructured data

Buy, Adapt or Build?

Data Considerations

2 important concepts:

1. You get a better ML model when you have more data
2. An ML model typically needs to be retrained for a new situation

Based on those concepts, here's (high-level) how you choose:

- Determine if the buyable model is solving the **same problem** that you want to solve. Has it been trained on the same input and on similar labels?
- Does the vendor have **more data** than you do?

When to Buy

When a credible service provider offers a turnkey service or API that solves your problem, you should seriously consider it. It'll almost always be less expensive and better maintained than something you build - a popular service will be able to amortize its development budget among hundreds of users, whereas you will have to pay for all improvements yourself if you build it yourself.

Moreover, if there's already a marketplace for that solution, it means the capability is not core to your business - your team can be doing something more distinctive and differentiated.

3 levels of abstraction at which you can buy AI capabilities:

- SaaS
- Building Blocks
- Enterprise Applications

What Can You Buy?

A cloud architect in a company that's adopting AI must become familiar with vendor choice and vendor management. You could save months of effort and dramatically lower risk of failure by choosing the right vendor.

Here are some capabilities (from late 2023 when authors wrote it):

- Better customer service
- Workflow assistance
- Improving marketing spend
- Writing sales and marketing content
- Recommendations
- Publicly available or gatherable data
- Retail
- Manufacturing
- Supply chain management
- Healthcare
- Media / entertainment
- Back-office operations
- Financial services

How Adapting Works

Take a SOTA model trained on an appropriately large dataset that is similar to yours (aka **pretrained model**, or **foundational model**) and then adapt that SOTA model to fit your data. Adapting is an in-between choice, between buying and building.

There are 2 ways to do this adaptation:

1. **Transfer learning** - keep the 99% of the model that has learned how to extract information from data and then train the remaining 1% of the model that operates on the extracted information to come up with the result
2. **Fine tuning** - This keeps the entire original model and tunes the entire thing on your data but keeps the magnitude of the weight updates small. Some **'low-rank'** generative AI fine-tuning methods train an ancillary model that adapts the weights of the foundational model

Hyperscaler ML platforms provide a fully managed experience for adapting pre-trained models.

AI Architectures

Authors list 7 types of AI apps that are currently successful:

- Understanding unstructured data
- Generating unstructured data
- Predicting outcomes - other attributes are the primary determinant of whether an event will happen
- Forecasting values - pattern of change over time is the important part - not whether an event will happen
- Anomaly detection

- Personalization
- Automation

Understanding Unstructured Data

Deep learning doesn't require feature engineering like traditional ML methods. That's why you can build a very generic deep learning system. For image detection, the same model **architecture** can be used to differentiate screws from nails as fractured bones from intact ones. Though they'd likely be different **model**s. Architecture is how the model is designed, and model is the actual weights of an instance of that architecture - you could train the same architecture on 2 different datasets and have 2 completely different models.

The research community has developed and published model architectures for standard problems in image, video, speech and text analysis. Anyone can use them, but you'd have to use/find your own data to train them. Therefore, for unstructured data related problems, usually it's not necessary to design your own model architecture - just pick a SOTA one off the shelf and train it on your data. All cloud providers provide simple ways to do this.

Generating Unstructured Data

Most GenAI use cases involve an LLM and prompting via an API. Crafting language for the prompt becomes important, and that's **prompt engineering**. Prompt engineering is good for prototypes, but version changes in the models can cause existing prompts to break. You can guard against this by fine-tuning your own model and keep it under your own control for production use cases. You can fine tune in this way on any of the public cloud platforms. You do it by training the LLM for a few epochs using a supervised parameter-efficient method (PEFT or SFT) such as Quantized Low-Rank Adaptation (QLoRA). You can do this with proprietary models too like OpenAI GPT-4 and Google Cloud PaLM through their platforms.

Instruction tuning can teach the model new tasks but not new knowledge (gotta re-train for that). But to do that cheaply, you can use:

- Few-shot learning - let the model learn from a few examples in the prompt
- RAG - let the model incorporate some given context like text or even images or audio to retrieve/ground answers to prompts
- Agents - generate structured data that can be passed into an external API

Predicting Outcomes

Historical data will serve as training data and it usually exists in logs and transactional DBs. Ideally it's easier to query the data from a DB. And the labels should be in the DB as one of the columns, ideally. Therefore, the architecture requires replication mechanisms and connectors to land the data in the DWH. Cloud providers (Redshift, BigQuery) provide ways to train models

directly on SQL data. Training should be relaunched anytime there is a sufficient amount of new data OR one of the features has drifted (this also requires the right observability).

Forecasting Values

Most ML modes do inference - they ‘predict’ the value of an unobserved field based on other factors - the key there is that the inputs do not include the field/value being predicted, even if you have actual historical outcomes/labels from the past.

In contrast, forecasting values involves having as part of the input previous actual historical outcomes/labels of the thing to be predicted. In essence forecasting problems require you to feedback the ground truth of past predictions as input for the next prediction. This real-time feedback loop makes the data engineering architecture considerably more complex.

The problem is you need to maintain 2 data pipelines: a batch pipeline for training based on historical data and a streaming pipeline for forecasting based on real-time data. However, a better approach is to use a framework that lets you use the same code for both batch and streaming datasets - Apache Beam - as this limits the exposure to training-serving skew.

Anomaly Detection

In anomaly detection on time-varying data, there are 2 time windows. Data over the longer time window (say a month) is clustered. Then, data over the shorter time window is compared with those clusters to find outliers.

The architecture here is similar to the forecasting values one except there’s 2 time windows that need to be maintained. For detecting individual anomalous events the principle is the same except that the time window is a single data point or session.

Personalization

For personalization models, for example, both the characteristics of the user (demographics) and the characteristics of the item are key features. The personalization model takes historical outcomes from similar people and ranks items based on which are most likely to be purchased/clicked/etc.

The cold-start problem comes into play here - with a new user who you don’t have data on, you still need to recommend something. Usually this is handled with a preprocessing step where it uses past users who we didn’t have data on and what they bought/clicked/etc. to recommend stuff.

Postprocessing is also important, for example, for filtering out items that the user has recently purchased - no need to show it again. Usually it’s a heuristic or set of rules for postprocessing.

For architecture, the model should be deployed as a pipeline. There's a preprocessing step where the most popular items are used to make a statistical model for solving the cold start problem, a postprocessing step is used to filter out results (here we need a real-time connection to storage that exposes recent purchases) and the recommendation model is used to rank items to be shown. If the list of items is too large, there can be a 2nd preprocessing step to rank items before the user even starts their session and save a smaller set of recommendations for each user.

Automation

Like personalization, automation requires preprocessing, postprocessing and multiple different models invoked as a DAG.

A small sample of high-confidence predictions is sent to a human expert for ongoing validation - **human over the loop.** In addition, decisions that are costly, low-confidence or difficult to reverse are also subject to human approval - **human in the loop**.

Human decisions on low-confidence predictions must be captured and treated as labels for subsequent retraining.

The set of models is tied into a pipeline and orchestrated in something like Kubeflow Pipelines for which managed services exist in the public cloud. The queue for low-confidence predictions needing human oversight could be any distributed tasks queue like SQS or Pub/Sub. The training data will be on blob storage or DWH depending on whether it's structured or unstructured.

Responsible AI

Because ML models are never perfect, you have to be careful about using ML in situations where decisions are made automatically without any human intervention.

AI Principles

Usually an organization will have some high level principles for when and where AI can be used/not used.

Example principle areas:

- Fairness
- reliability/safety
- privacy/security (e.g. use **federated learning** so users data is never sent over the internet)
- Inclusiveness

- Transparency
- Accountability (e.g. decisions should be accountable to people)

ML Fairness

Be careful about using historical data in training ML models, especially when they make irreversible decisions that affect people, because the ML model will pick up the biases inherent in that data.

There are many tools that are part of cloud ML frameworks to help with diagnosing ML fairness issues. For example:

- sliced evaluations - fairness testing technique where you evaluate your ML model's performance separately across different subgroups or "slices" of your data—essentially breaking down your test set by demographic categories to spot disparities
- what-if tests (these are integrated into Vertex AI Workbench) - experiments that simulate hypothetical scenarios (inputs) to assess how ML models behave when sensitive features/inputs are altered
- Continuous model monitoring on protected criteria

Explainability

Often stakeholders (end users, regulators) will refuse to adopt an ML model unless they get some indication of why the model is making the recommendation. Cloud ML frameworks provide explainable AI (XAI) tools that can help you understand how your ML models make decisions.

Chapter 11

Architecting an ML Platform

The previous chapter was about architecture of ML apps, often using prebuilt models. In this chapter we delve into the development of **custom models**, including stages of development and frameworks that support them. We'll look at how to automate the training process using tools and how to monitor the behavior of trained models that have been deployed to endpoints to check for drift. In previous chapters we've talked about ML capabilities enabled by various parts of the data platform: data storage in data lake or DWH, training on compute that is efficient for that storage and inference that can be invoked from a streaming pipeline or deployed to the edge - here we'll pull all these discussions together.

ML Activities

What activities does an ML platform need to support?

The process usually goes something like this:

- Clean and process the raw data
- Exploratory data analysis
- train/test split the data
- train and test the model (iterative)
    - Train
    - Test
    - Check for compliance and performance
    - Deploy
    - Clients request inference
- Automate the steps for training and deployment
- Continuously monitor, evaluate and retrain

Developing ML Models

This consists of iteratively:

- Preparing the data for ML
- Writing the ML model code
- Running the ML model code on the prepared data

Labeling Environment

In some cases, labels are naturally present in the data, and in others, you'll need to get the data labeled by human experts. Quite often this tooling is outsourced.

Development Environment

Because the ML development process is so iterative, data scientists are most productive in an environment where they can write code, run it, view the results, change the code immediately, and rerun the code, ideally in small snippets.

Use Jupyter notebooks.

Notebook server runs the code and is installed on cloud VMs - this service is offered by all hyperscalers, and cloud-agnostic data frameworks like Databricks.

There exist domain-specific managed notebooks like terra.bio.

User Environment

Typically notebooks are user-managed - treat the VM where the notebook server runs as the DSs workstation - they'll spend most of their workday in it.

Often DSs work with confidential data or PII, so it's common to situate the servers and data behind a higher trust boundary like a VPC.

Make sure to use column/row level security when providing access for data scientists.

Data scientists require VMs with GPUs when training deep learning models on unstructured data. Cost can become an issue, so there are many solutions for that depending on the need. For one, you can find and stop unused instances, but for large teams (>50 DSs), that's not enough. You may want to amortize computational power across users by having them share compute resources. In such cases, the architecture involves running the notebook servers on Kubernetes cluster rather than on a VM (JupyterHub) and then serving out notebooks to multiple users from the same cluster. Because this architecture is multitenant, the tradeoff is security risk and/or complexity from solving the security risk - the cost savings may or may not be worth the risk.

Preparing Data

The first step in an ML project is usually to extract data from source systems (DWH maybe), preprocess/clean it, and convert it to a format (TF records for example) optimized for ML training. The resulting ML data is stored in an object store for easy access from notebooks. For example, image classification models often need data rotated/flipped at multiple degrees or for all images to be the same size.

Then DSs will visually examine the data, often using common plotting libraries to understand the distribution. During examination, the DS might notice issues that require correction/discarding of data - this can be done by updating the preprocess code OR just add the correction/change to the software loading the data.

Often this can all be done with SQL, but sometimes DSs need something more powerful, so they'll use a programming language and libraries like python and pandas for small datasets, or Dask/Spark/Beam for larger amounts of data. Again, this data and processing needs to be within the high trust boundary.

Always make sure these activities are carried out in coordination with the business to have a clear understanding of the organizations objectives.

Training ML Models

Your DSs will write model code with Jupyter notebooks in frameworks such as scikit-learn, XGBoost, Keras/Tensorflow, or Pytorch, and then they'll execute the code over the training dataset.

Writing ML code

The code will typically read training data in small batches and run through a training loop that consists of adjusting weights. The data pipeline that reads the data might carry out data augmentation to artificially increase the size of the dataset by, for example, flipping images. The model architecture will be closely tied to the inputs that are read and how they are transformed. The data scientist will typically experiment with many options.

Model code is developed iteratively, and so a low-latency connection to the cloud and the ability to have a quick turnaround time are essential. It is important to try out small changes to the code without having to run the entire training program again. Writing code is typically *not* the time to use managed training services - at this stage, data scientists (DSs) use notebooks in their local environment.

Once the code has been developed, DSs will typically want to run the training job on the entire dataset.

Small Scale Jobs

For small jobs, it's useful to just run the training in the notebook itself.

Managed notebook services offer DSs the ability to change the machine type on which their notebook runs, to make it more powerful - like switching to a GPU or TPU machine. That allows faster training for moderately sized datasets.

For even larger datasets and/or complex models or where a training run can take longer than an hour, it is preferable to use a managed training service. It is possible to submit a notebook to a managed training service like SageMaker or Vertex AI, and the notebook get executed in its entirety.

Distributed Training

For extremely large datasets, scaling up isn't enough - you must scale out using a cluster of machines with special communication hardware/software. Frameworks like pytorch and tensorflow support distributed training, but the cluster must be setup in a specific way.

No-code ML

Custom models do not always require coding in TF/Pytorch/etc. Low-code and no-code options exist. Use, for example, AutoML and tools like Daitaku and DataRobot to train and deploy with clicks.

You can treat the model that comes out of AutoML as a custom model that can then be deployed just like models that were coded up by a DS team.

Deploying ML Models

You can do batch inference or online inference.

If you're only doing batch, you could potentially just load the model file into Spark/Beam/Flink - no need to deploy. But for online inference, you need to deploy it as a microservice somewhere.

Deploying to an Endpoint

Clients access an endpoint through the URL associated with it - they sent an HTTP POST request with a JSON payload containing the input to the prediction method. The endpoint contains many model targets, among which it splits traffic. The model is an object that references ML models built in a wide variety of frameworks (TF, PT, XGBoost). There are prebuilt container images for each framework.

The endpoint is backed by an auto-scaling service. The computation can be sped up by using accelerators. The machine may need to have enough memory. Also to note is that auto-scaling can introduce latencies (cold start problem) - so solve that with a warm pool and/or faster image pulls.

Evaluate Model

The ML Platform will need to allow for A/B testing different model versions so ML engineers can decide when to replace/retrain models in prod.

The ML Managed services provide the ability to monitor resource usage and ensure that the deployed model can keep up with the input requests. A sample of the inputs and corresponding predictions can be sent to a DWH and used to compare the performance of different model versions.

Hybrid and Multicloud

Because moving data around is costly and adds governance and security considerations, you will usually choose to train ML models on the cloud where most of your historical data lives. OTOH to minimize latencies, you'll need to deploy to the cloud or to the edge where consuming

apps run. To do such hybrid training and deployment, use standard model formats (TF SavedModel, ONNX, .bst) and containers.

This flexibility to run inference disconnected from the cloud, is an important consideration when building an ML platform. Choose frameworks that are not tied to proprietary implementations.

Training-Serving Skew

When an ML model is trained on preprocessed data and then used for inference, it's important to carry out the same steps on each incoming request as those carried out for preprocessing training data - otherwise the inference requests will be seeing different data than what was seeing during training, and you'll get a skew.

There are 3 ways to ensure preprocessing done during training is repeated as-is during prediction:

1. Put the preprocessing code within the model
2. Use a transform function
3. Use a feature store

Within the model

This is the simplest option, and that's the biggest reason to use it where possible. Preprocessed code is carried along with the model, so deploying on the edge or in another cloud results in the same results. The SavedModel format contains all the necessary information.

The drawback here is that the preprocessing steps will be wastefully repeated on each iteration through the training dataset. The more expensive the computation, the bigger impact of this drawback.

A second drawback is that data scientists have to implement the preprocessing code in the same framework as the ML model - that can be hard/cumbersome, especially if the preprocessing code uses custom libraries.

Transform function

Capture the preprocessing steps in a function and use it to preprocess data before actually feeding the preprocessed data to the model during training - that way you're not processing the data repeatedly during each epoch; that preprocessed data can be reused. Of course the preprocess function/steps need to also be run during inference. While this adds efficiency, it also adds complexity, which is the drawback.

If there are feature(s) that are computational expensive to compute, the extra infrastructural and bookkeeping overhead may be worth it.

Feature store

Feature store is a repository for storing and serving ML features. It's essentially a kv-store where the key consists of an entity (e.g. hotel_id) and a timestamp and the value consists of the properties of that entity (e.g. price, number of bookings, number of website visitors to hotel listing over the past hour, etc.) as of that timestamp.

The first situation you'd need a feature store is if the feature values will not be known by clients requesting predictions/inference but have to instead be computed on the server.

The second situation is to prevent unnecessary copies of the data. For example, consider that you have a feature that is computationally expensive and is used in multiple ML models.

Don't go overboard in either scenario. For example, if the features are all computed in the same way and/or there's not too many, it's fine to use a preprocessing function. Similarly, if there's not a lot of repeated computation to compute features and the transform/preprocess logic is simple, it's fine to just include it in the model.

The canonical use of a feature store

The most important use case of a feature store is when situations #1 and #2 both apply. Consider a feature that is used by many different models but is also continually improved and computationally expensive. For example, embeddings of a song/artist/user in a music streaming service that get recomputed daily. The training code will need to fetch the values of this feature that align with the training labels and the latest version of the embeddings. This has to be done efficiently and easily across all labels. And this has to be done across the tens or hundreds or thousands of features used by the model. The feature store makes periodic model retraining on hard-to-compute, frequently updated features exceptionally useful.

Decision chart

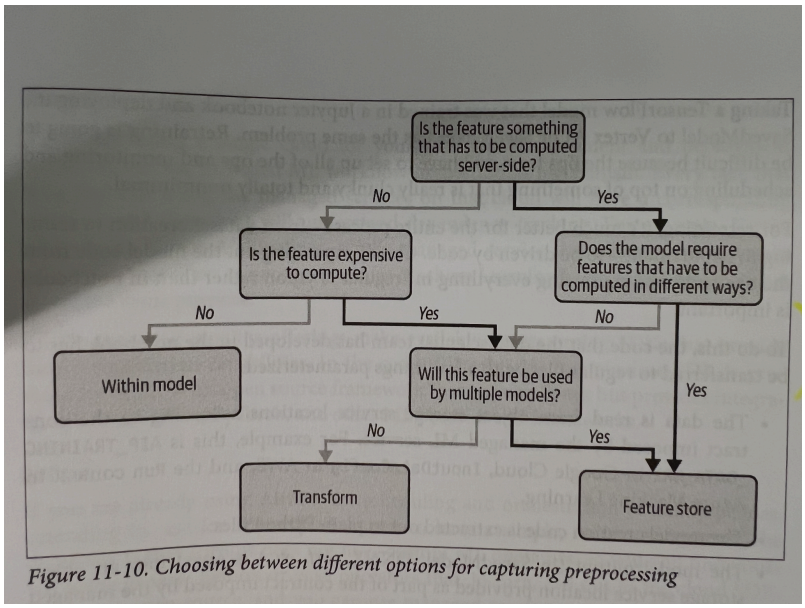

Figure 11-10. Choosing between different options for capturing preprocessing

Automation

Though you can deploy a custom model using cloud consoles or cloud managed notebooks, neither of those approaches scale to hundreds of models and large teams.

Automate Training and Deployment

Retraining a model that had been trained and deployed via cloud consoles is not easy - you may have to wake up at 2am, use a web UI and retrain and deploy. It'd be better if it all was automated via code.

To automate the entire process (from dataset creation to training to deployment), you need to separate model code from ops code and express everything in 'regular' python rather than notebooks.

To do that, the code that the data scientist team has developed in notebooks has to be transferred to regular python files and a few things parameterized:

- The data is read from object storage - you need a pointer to it
- The model creation code, extracted out to python files
- Model output (e.g. SavedModel, ONNX, .bst, etc.) saved to object storage

Orchestration with Pipelines

You must orchestrate the steps on the right infra:

- Prepare the dataset (Spark, Beam, SQL, etc.)

- Submit a training job to the managed ML Service
- If the model does better than previously trained models, deploy it
- Monitor the model performance and do A/B testing

There are, broadly, 4 options for orchestration:

1. Managed pipelines
2. Airflow
3. Kubeflow
4. TFX

Managed pipelines

If you're entirely on top of a single cloud, use the managed pipeline offered by it (e.g. Vertex AI Pipelines for GCP). Another example is MLflow if you're on Databricks. It runs on Spark but integrates with TF, PT and other frameworks.

Airflow

If you're already on airflow (open source) for scheduling & orchestration, extending it makes a simple, consistent system. The drawback is that airflow doesn't have custom ML operators, so you'll be writing python or bash operators that call out to the cloud SDK.

Kubeflow Pipelines

Each of the steps in your pipeline can be containerized and the containers orchestrated on K8s. Kubeflow operators already exist for popular ML frameworks, ML metrics, etc. and it's open source. Lastly, there are cloud-agnostic managed Kubeflow providers like Iguazio.

TensorFlow Extended

If you're using the TensorFlow ecosystem, TFX is a good choice, offering convenient python functions for common operations like model validation, evaluation, etc. It avoids the need to build and maintain operators or containers. TFX is an open source package that can be run on managed pipelines, Airflow and Kubeflow.

Key libraries in the TFX ecosystem:

- TensorFlow Data Validation (TFDV) - detecting anomalies in the data
- TensorFlow Transform (TFT) - data preprocessing & feature engineering
- Keras - building and training ML models
- TensorFlow Model Analysis (TFMA) - model evaluation & analysis
- TensorFlow Serving (TFServing) - serving ML models to endpoints

Continuous Evaluation and Training

Each step in the ML workflow outputs artifacts which must be systematically organized.

Artifacts

Examples include:

- User notebooks, pipeline source code, data processing functions, model code
- Parameters, metrics, dataset references, model references
- Model training jobs, trained model files, deployed models
- The environment used for development, training, deployment in a container (in a container registry)

Dependency tracking

Steps in the ML pipeline should do dependency tracking and not launch the task unless one of the dependencies has changed. You can setup a CI/CD pipeline including continuous evaluation and continuous retraining.

Continuous evaluation

There are a few steps necessary to carry out continuous evaluation:

- Prediction endpoints store a sample of inputs and corresponding predictions/outputs
- Humans are notified/ticketed and validate the predictions periodically
- Scheduled evaluation query to:
    - Compare the human labels against predictions to monitor for **drift (label, data)**
    - Monitor distribution of features for **feature drift**

Continuous retraining

If the evaluation query identifies skew or drift, an alert should be raised. Sometimes a human will have to examine what types of changes are needed to the model, and sometimes an automatic retraining can be triggered.

Other reasons to kickoff a retraining:

- the model code changes
- A sufficiently large amount of new data is received

Choosing the ML Framework

As a cloud architect, you should choose the tools available in the ML platform based on the skill set of the people who will be building the ML application. Some examples:

- Data scientists need a code-first ML framework like Keras/TF/PT and operations with SageMaker/VertexAI/Databricks

- Domain experts need a low-code ML framework like BigQuery ML or AI Builder
- Practitioners in the field need no-code ML framework like DataRobot or Dataiku

Team Skills

The answer for the same ML problem will vary from org to org. For example, consider using ML for pricing optimization:

- A team of DSs will choose to build a dynamic pricing model
- A team of domain experts might choose to build a tiered pricing model
- A team of nontechnical practitioners determines the optimum price themselves

The right answer depends on the people/skills available, the business and the ROI that leadership expects.

Task Considerations

The end-to-end ML workflow consists of:

1. Training
2. Deployment
3. Evaluation
4. Invocation
5. Monitoring
6. Experimentation
7. Explanation
8. Troubleshooting
9. Auditing

As a cloud architect, you need to identify who at your company will be doing the task for each ML problem. Very rarely will all the steps be carried out by an engineer (authors say that's actually a red flag if they are).

For example, let's say we need a model that'll be invoked as part of reports that are run within Salesforce by a practitioner (a salesperson, in this case) and audited by a Sales Manager - the model will be build by a Data Scientist. If the DS builds a model in Spark, productionizes via a notebook and deploys it as an API, how will the Sales Manager do their audit? Then you need another team to build a UI around it - huge waste of time/money/effort. A better choice for a cloud architect to make would be to deploy the model into a DWH that readily supports dashboards.

User-Centric

Organizations that standardize on an ML framework disregarding the skill set of the people who need to carry out a task will fail.

Make sure to choose open, interoperable tools so that you don't end up with silos.

Chapter 12

Data Platform Modernization: Model Case

In this final chapter we'll apply the principles from all preceding chapters to explain what it means to transform an old-fashioned data platform into something that is modern and cloud native.

New Technology for a New Era

YouNetwork is a successful tech company that sells cable/internet. Their customers have STBs (set top boxes) that they use to access services, and the company innovates in content. There's been a recent explosion of data generated by new services and the company hasn't been able to act with agility based on the real-time data caused leadership to reflect of the future of its tech stack.

The main drivers:

1. Scalability of the services with increasing data volumes
2. Necessity to quickly implement and put into production new analytics-based solutions
3. The need to offer more tailored content streams to customer who expect more personalization

The Need for Change

Traditionally, YouNetwork would invest in new tech by building capability in its own data centers, but scalability issues made this harder.

YouNetwork has gone through 2 digital transformations in its history:

- **COTS Era (Commercial off-the-shelf)**
- **OSS Era (Open source software)**

The board recognized the need for a 3rd one to the public cloud.

First decision was to create a technical 'SWAT' team consisting of internal leaders and contractors from consulting firms to prepare a solid business plan to modernize YouNetwork's current data platform.

It Is Not Only a Matter of Technology

SWAT team found it's important to align cloud strategy with overall organization strategy to consider how the business would evolve with the technological update - e.g. more data faster, enabling better content decisions, resulting in content acquisition team actually being a stakeholder.

There was a need for sponsorship at the top, which they already had, and that sponsorship needed to flow down through the chain of the company. They attacked that problem with creation of Cloud Center of Excellence (CCoE) to identify people who could carry out that transformation. YouNetwork used both internal (engineering, product, architecture) and external (senior consultants with expertise and experience) people to join the CCoE.

Main question: **"How do we guid teh company into a new chapter?"**. The pivotal task was to carefully analyze, compare and judge the proposals made by cloud vendors to select the best one that fit the organization's needs.

CCoE clarified the process to be used:

1. Governance - identify and implement standards
2. Migration management - push and control the migration of workloads to the cloud
3. Operationalization - make the cloud environment workloads operable
4. Training and support - serve as a SME for eng/product/ops to foster further adoption

The Beginning of the Journey

YouNetwork had to first identify a possible target architecture an then define how to make it real considering its on-prem legacy heritage.

The Current Environment

The company decided to issue an RFP to the various cloud vendors.

The legacy system:

- STBs send chunks of data (CSV, JSON, Parquet, etc.) every 15 minutes to an FTP server that stores them
- COTS ETL solution reads/ingests data from the FTP server to the DWH and data lake via Hadoop/Spark tech
- Comms between Hadoop cluster and the DWH done via an ad-hoc connector
- Data analysts/decision makers carry out their work via Excel and other BI tools reading from DWH

Hardware was close to EOL, software licenses were very expensive and software was also out of date. The architecture meant streaming patterns couldn't be adopted.

The system resided in 2 data centers - main and failover sites. Configured in active/active. DWH was synced in real-time, achieving recovery point objective (RPO) close to zero. Data lake data was aligned multiple times a dayusing DistCp with a target RPO of 6 hours. In case of disaster, it was possible to route the traffic from one data center to the other one with a minimum downtime achieving a recovery time objective (RTO) near zero.

The Target Environment

The goal was to have 1 single system (lakehouse) as a single point of truth for the entire company. All users should be able to get access to the data, based on their access rights.

The goal of adopting OSS was to terminate as many of the COTS licenses as possible. The team chose a **SQL-first lakehouse** because the number of people who were familiar with SQL was higher than the number of users able to play with Spark.

Since the current architecture had scalability limits/issues, they decided to use a cloud solution with unlimited scalability and the ability to scale up/down to save costs.

Another key element was to provide real-time streaming data capability to provide customers more immediate and tailored feedback/content/recommendations/etc.

Furthermore, the high number of tasks to be managed and controlled meant a clear need for a workflow management platform layer. And every component need fine-grained control policies to preserve security/privacy.

Finally, the platform had to be the home for innovation by enabling and expanding the adoption of ML.

Authors say the difference between a successful vendor choice and a poor one is alignment between your business goals and the vendor's products.

There were 3 main motivations behind the renovation of YouNetwork's tech stack:

1. Grow its market share (scalability)
2. Offer more attractive services (streaming)
3. Develop fidelity (personalization)

The PoC Use Case

YouNetwork gave cloud vendors a specific problem with 2 motivations:

- Space needed to accomplish the task
- Time requested to get to the final results

Remember the current environment (STBs, FTP server, ETL, DWH, visualization tool). The major drawbacks were:

- Inability to scale
- Size of the requested space
- Time to access the data

Note: always choose a use case that's harder than typical, has significant business impact, and can shed light on the relative benefits of a move to cloud. Talk to business users when selecting a PoC use case.

The RFP Responses Proposed by Cloud Vendors

The Target Environment

Main components of the architecture:

- Data sources
    - Real time
    - Batch
- Ingestion
- Transformation
- Analytics
- BI and AI/ML
- Data consumers

Horizontal services on top of all that:

- Orchestration
- Catalog and privacy
- IAM/operations/logging and monitoring
- Continuous integration and continuous delivery

In terms of development, high level of flexibility options for the company:

- IaaS - current environment is most like this
- PaaS
- SaaS

Though current environment was like IaaS, to achieve all the benefits of cloud, YouNetwork needed to move towards PaaS/SaaS.

The Approach on Migration

"How do we get there?"

Vendor proposals converged on a multi-step approach based on 4 phases:

1. Foundations development (landing zone)
2. Quick wins migration (find workloads that easily fit)

3. Migration fulfillment (migrate all the data for those workloads, e.g. historical STB data)
4. Modernization (start adding new capability that's been unlocked)

Foundations development

This refers to baseline and structural configuration of a cloud environment - the minimum configuration to put in place to enable migration from legacy. Generally includes all horizontal services that make the platform nice for the workloads.

- Identity management
    - Cloud identity services
    - Auth options
    - Auditing
- Access management
- Security
- Networking
- Visibility

Quick wins migration

For PoC, Identified use case of data collector from STB datathat enabled tech users to have a better understanding of the status of the footprint of devices. Quick win: enable system to collect metrics from multiple STBs, leveraging a full serverless pipeline.

Migration fulfillment

Focuses on **speed of migration**, so suggested approach was to **rehost** with a **replatform** nuance when possible. This allows the org to:

- Define a standard for the various migration activities
- Automate migration activities

Data migration included:

- Hot data
- Cold data

Modernization

Important aspect: every single component of the platform has to be accessible via APIs - it is the core element of the transformation because it enables integrations between internal and external resources, future reorganization of the architecture, and real-time interoperability between components. Main transformation for each layer:

Ingestion - get rid of the FTP server approach (slow, not scalable, insecure), use a layer handling data at scale in real time using message queuing/event paradigm. Data must be stored maybe in original format in blob storage.

Transformation - start adding open source and removing licenses

Analytics - use serverless, fully managed solutions to achieve scalability

BI and AI/ML - Personalization, content optimization, predictive maintenance, audience analytics, fraud detection

The RFP Evaluation Process

YouNetwork selected a subset of the vendors to attempt to build a PoC.

The Scope of the PoC

Cloud vendors had to implement a mockup - YouNetwork specified 3 KPIs for the PoC:

- 20M devices - a fleet of mock STBs (VMs)
- 2-week (nice to have: 1 month) window - extend size of historical data stored (cloud blob store)
- Less than 5 minutes - latency to materialize the table (Redshift/Synapse/BigQuery)

The vendors were given 2 weeks.

Both vendor participants approached in similar way:

- Design and configure a blob storage reachable via APIs from STBs
- Mimic behavior of multiple STBs
    - Generate random data based YouNetwork STB file example
    - Connect to backend to send data
    - Send data
    - Destroy itself after success response
- Launch STB mocks on VMs
- Develop a batch ETL serverless pipeline to read and transform all the data at scale and ingest it into the DWH
- Implement the dashboard

The Final Decision

Final decision had these considerations:

- Technical robustness - scalability, elasticity, availability, disaster recovery, backups
- Vision and roadmap
- Team and organization
- Partnerships

- Training programs
- Overall pricing

Peroration

When every stakeholder in your business has access to the data they need, they can make better decisions and solve problems more effectively. Making the use of data easy and widespread will give you a competitive advantage, help make better decisions, improve your products and services, and grow your business. You also need to ensure everyone has the skills and tools they need to access and use data, and you need to create policies that protect privacy and security.