Software Engineering for AI-Enabled Systems

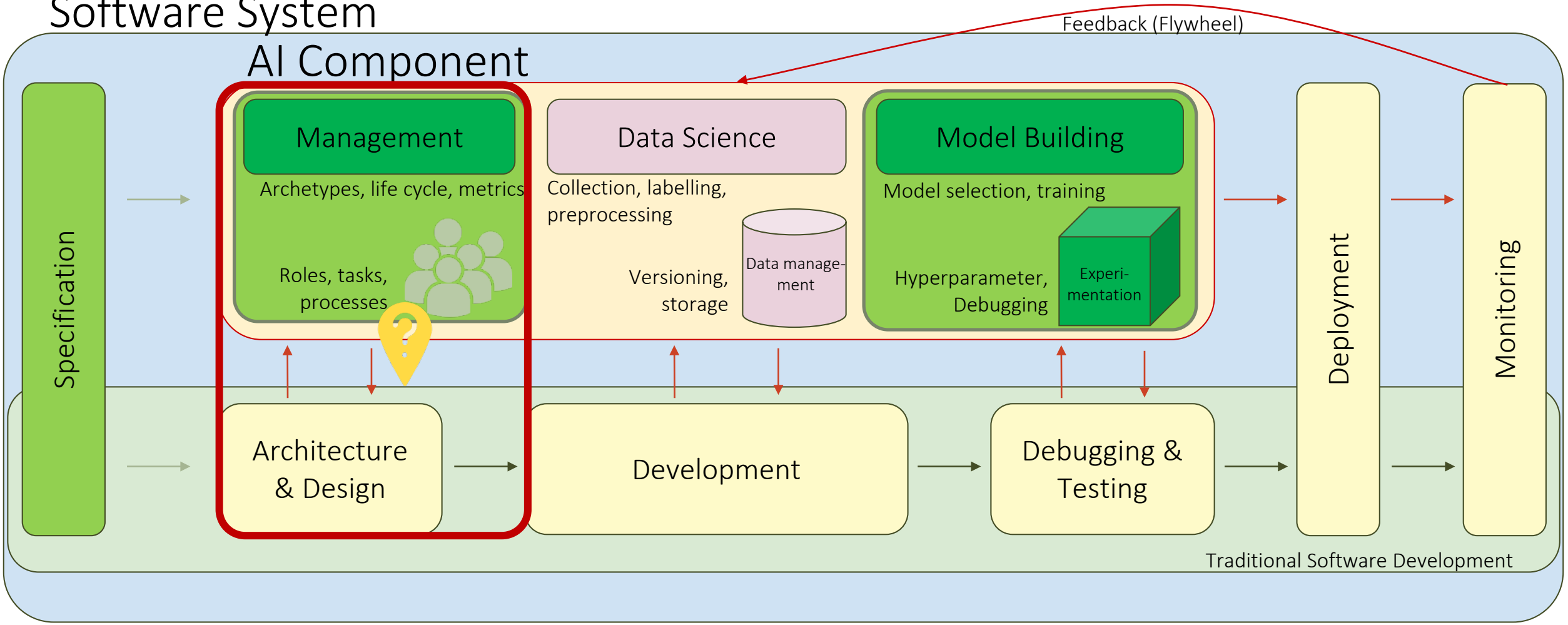
SOFTWARE SYSTEME

UNIVERSITÄT LEIPZIG

Prof. Dr.-Ing. Norbert Siegmund
Software Systems

Software System

AI Component



How to incorporate the AI component into the software system? What is a proper architecture for an ML component?

- How to manage information flow?
- How to bridge programming languages and system boundaries?
- How to design the system with modularity and software quality into account?
- SaaS / SaaFunction called by the system vs. embedded into the application vs. device deployments (embedded, mobile, server, etc.)

Topic 0:
General Development Process

TL;DR:

- Architecture diagrams (e.g., C4) support structuring the AI-enabled system
- Elements of an architecture comprise different types of pipelines, model server, feature server, parameter server
- Existing tool landscape needs to be considered to design an appropriate system
- Different architecture styles impact non-functional properties
- Best practices for designing a system
- UX-ML design principles

“This may come as a shock to some, but machine learning development is not done when you achieve a good score on your test set. If anything that gives you a good starting point for a production system, but as soon as the model encounters real-world data, that’s when development begins. “ Jimmy Whitaker (Developer Advocate @ pachyderm.io)

What constitutes an AI System?

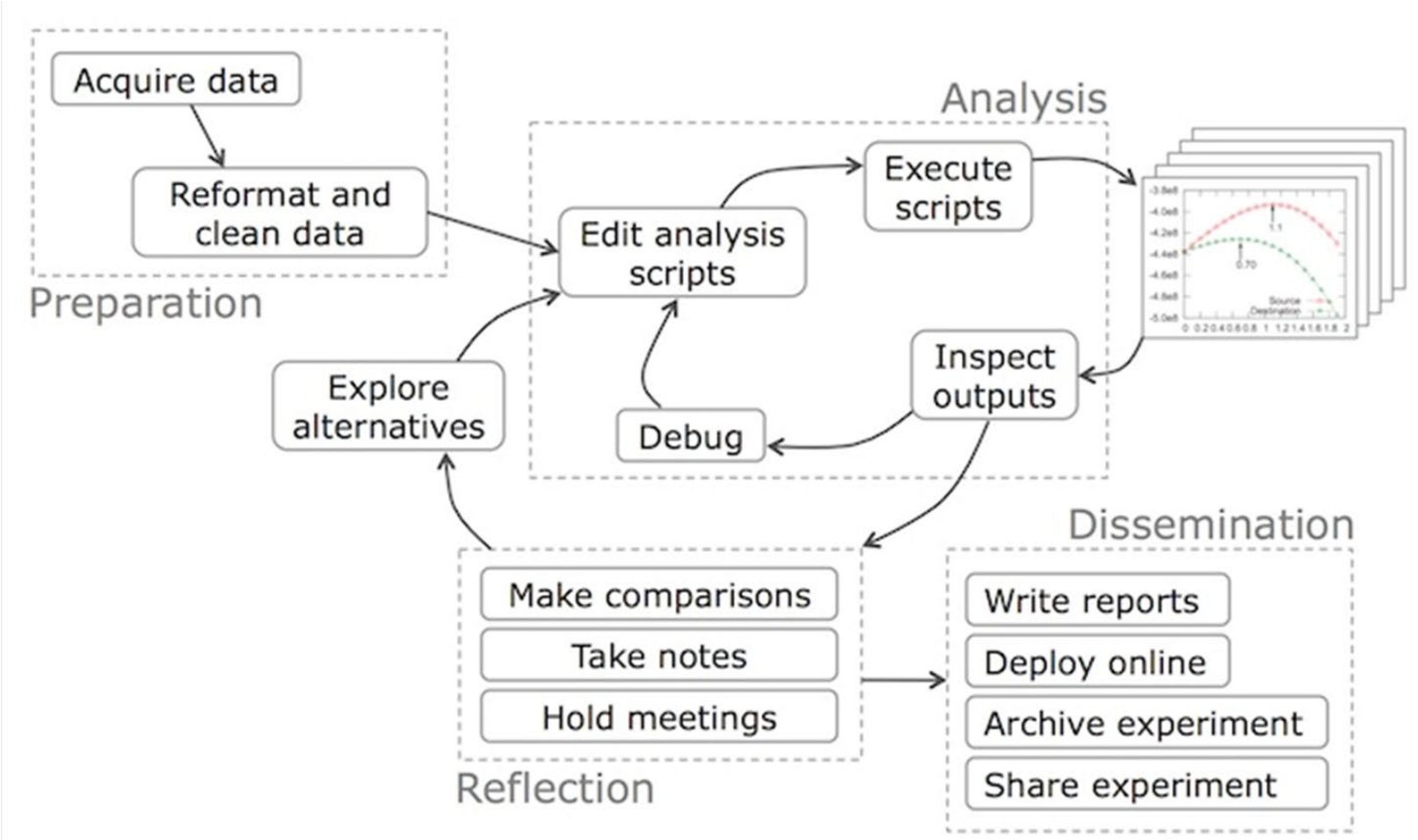
Data collection, data manipulation, data cleaning, data preparation, data labelling, feature engineering
(use of data lakes, labelling tools, exploration libraries)

AI system = Data + Code

Algorithm, model, glue code
(use of ML libraries, such as sklearn, TensorFlow, PyTorch)

Focus of most courses, academic research, and competitions

Workflow of Data Scientists: Iterative



https://cacm.acm.org/blogs/blog-cacm/169199-data-science-workflow-overview-and-challenges/

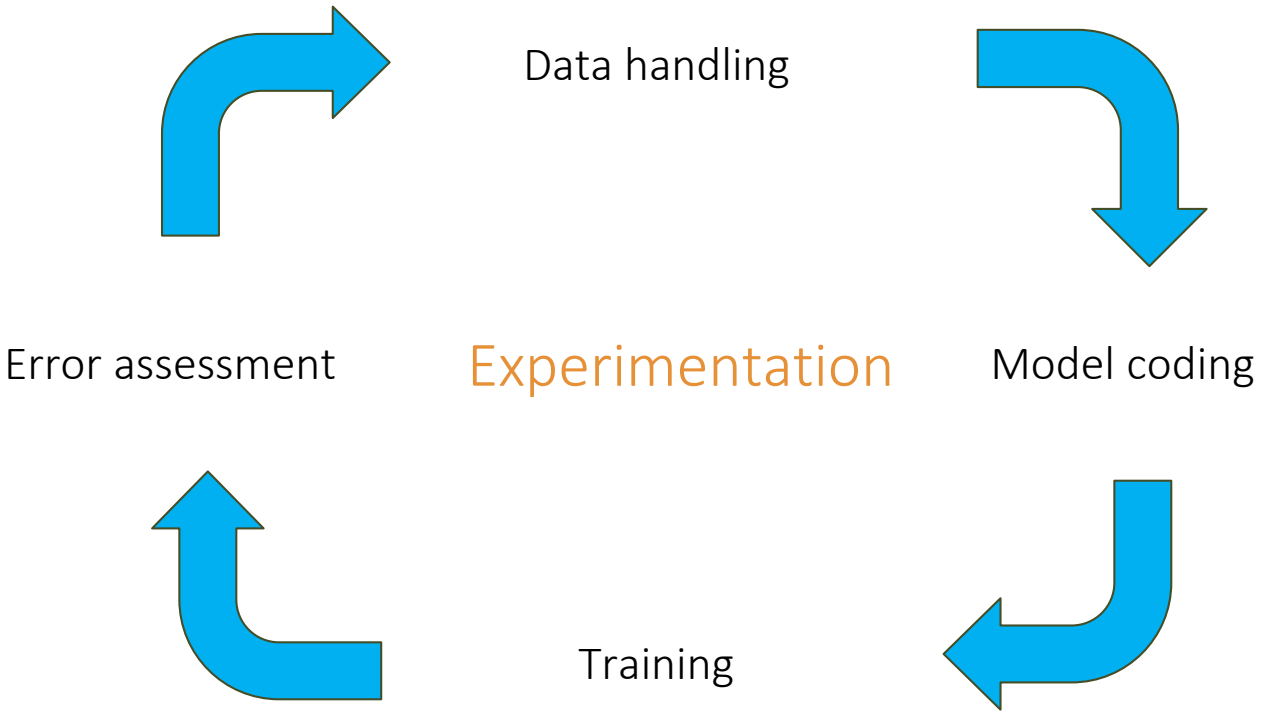
Iteration Goals

1. Improve and reach good training error
2. Reach similar results on an evaluation / test set
3. Measure and verify business goals and software metrics are met (user response, performance, throughput, etc.)

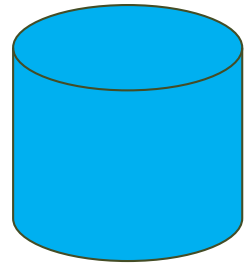
Always keep validity, fairness, and bias in mind!

Average error is useless if we get the important cases wrong! (cf. Model Testing slides)

Pipelining the Iterative Development

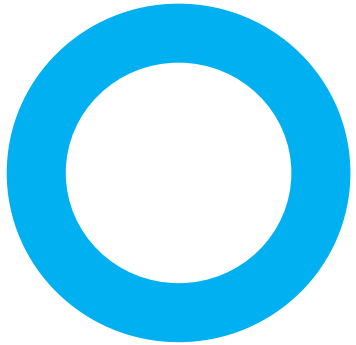


Experiment framework



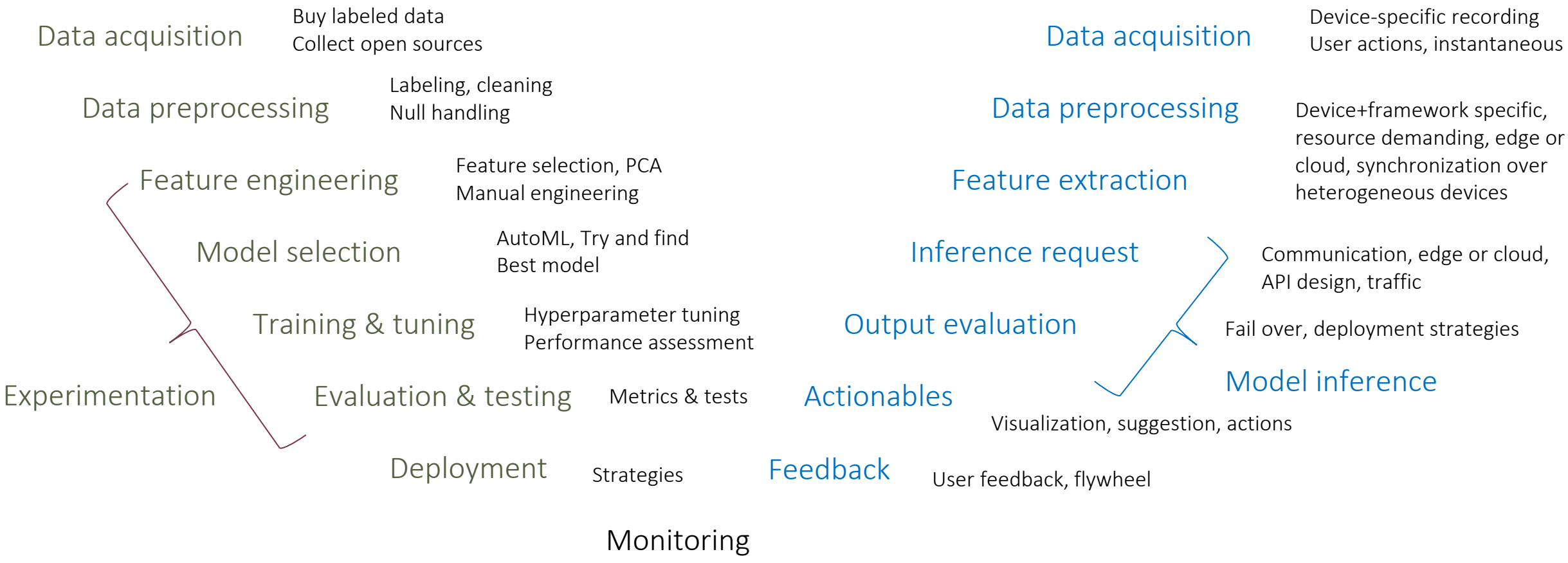
Track runs, analyze settings, tune hyperparameters, select new settings, reproduce results

Experiment pipeline



Automate data preprocessing, training, and error assessment; version model code; define and keep environment

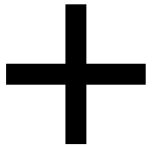
Training Pipeline vs. Inference/Production Pipeline



Inference/Production Pipeline Examples

Voice activation

Speech recording



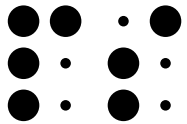
Sample homogenisation

API request

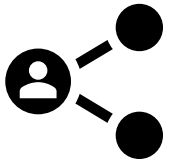
Play received sample

How to automate this?

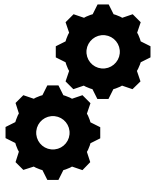
Clickstream of user



User profile

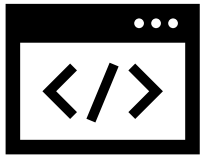


Recommender request



Query product catalogue

Show result



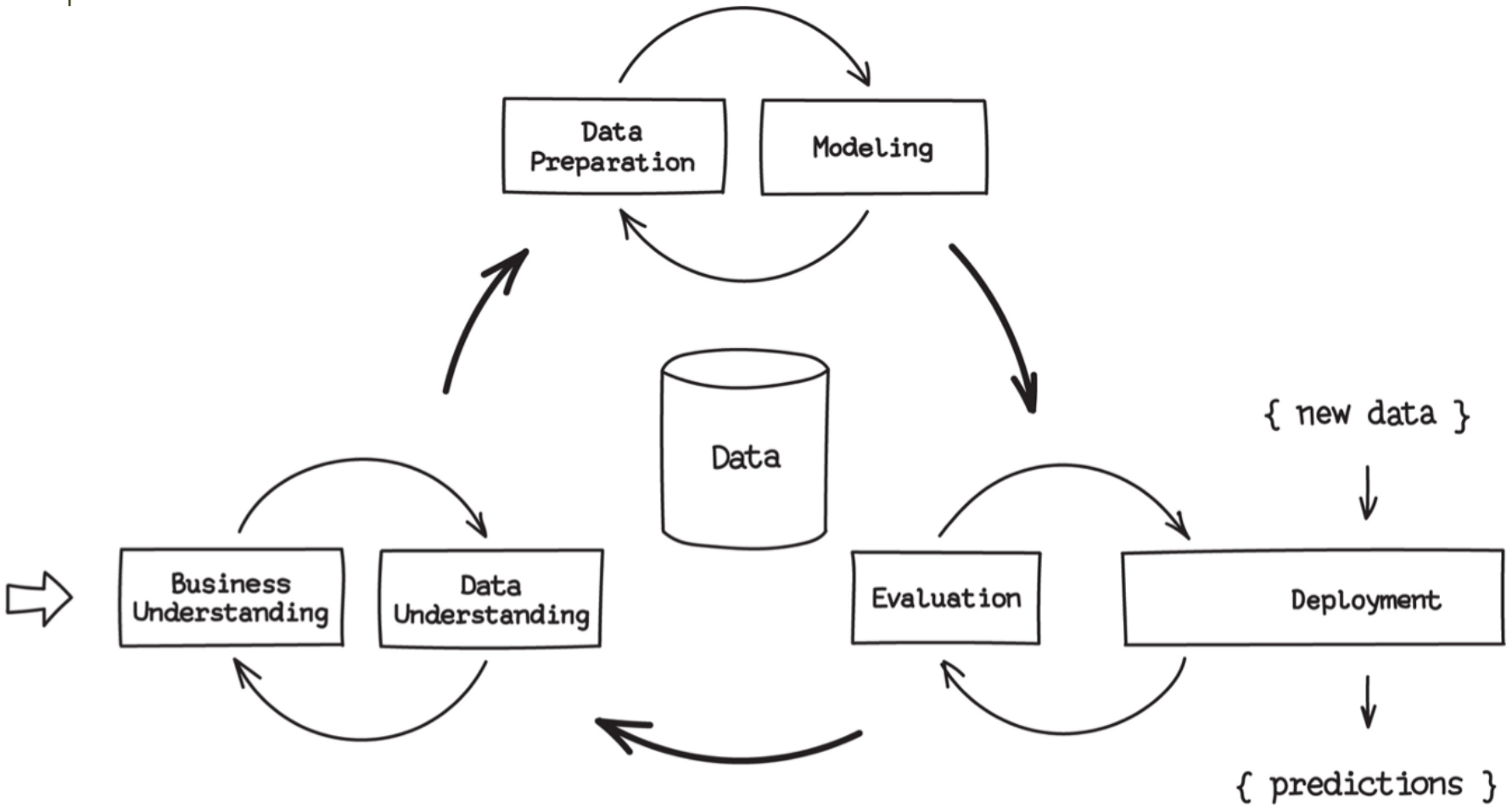
Experimentation: Overview

Developing an ML model means doing (scientific) experimentation involving a substantial amount of empirical work.

Center of the development process.

Two approaches:

- Blindly training
- Systematic training



Experimentation (when properly done)

Enables to:

- Decide when the goals (i.e., predefined metrics) have been reached and the model is ready for production
- Test for bias, unusual scenarios, assumptions in data and hardware
- Reproduce models, version the experiment
- Apply a root-cause analysis of false predictions

Source: https://towardsdatascience.com/a-quick-guide-to-managing-machine-learning-experiments-af84da6b060b

Experiment Tracking, Logging, and Analysis

Plenty of choices throughout the ML pipeline, but which choice leads to better solutions? What is the root-cause of bad score in experiment D and good score in E? Is it the data slice? The pre-processing mechanism? The hyperparameter? The choice of the model?

Empirical methods allows us to approach and answer these questions in a systematic way by exploring all decisions.

Systematic search requires to track what has been changed, what is new, what is removed, and what are the consequences of these actions on the recorded metrics.

Experimentation: Requirements and Gains

We need logging/tracking (at different places in the pipeline) of an experiment, which includes a concept of experimentation where we can group

- All relevant data
- Code, and
- Decisions for one specific run of the ML pipeline.

This must be reproducible, that is, multiple runs should produce the same result

- Random factors are either not allowed (use seeds) or include (many) repetitions + averages if necessary,
- Be independent of hardware or environmental influences.

Compare different experiments with each other to make sense out of choices and gradually improve your pipeline.

Topic I:
Software/AI Architecture

What is Software Architecture?

Many definitions… such as *which components exist at high level and how they interact*… or *all design decisions that need to be made in the project*.

Ralph Johnson: **“Architecture is about the important stuff. Whatever that is”**

So, what is important? It depends on the project:

- External services, infrastructure, modularization of the system, ways of communication, logging and monitoring capabilities, extendability, scalability
- Part of the SW architecture refers structuring the system according to a style (e.g., microservice, layered, pipes and filter).

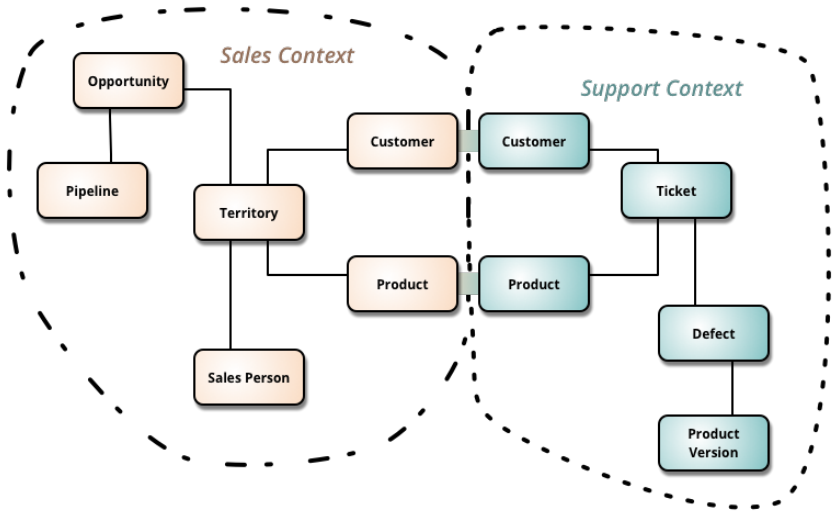
Process of Defining an Architecture

Different approaches exist (e.g., responsibility-driven design for OO systems; domain-driven design for distributed / microservice systems; clean architecture for enterprise systems).

General guidelines apply:

- Understand your requirements (functional constitute business logic, non-functional constraints on technological choices and communication layout)

- Understand your domain boundaries (what belongs to the same domain functionality – Fachlichkeit--, same language; team&data)

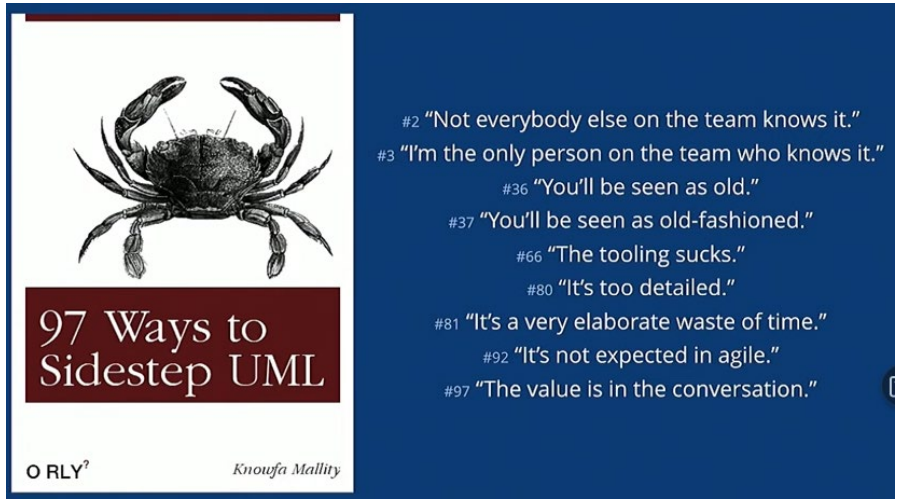


Different Views on the Architecture

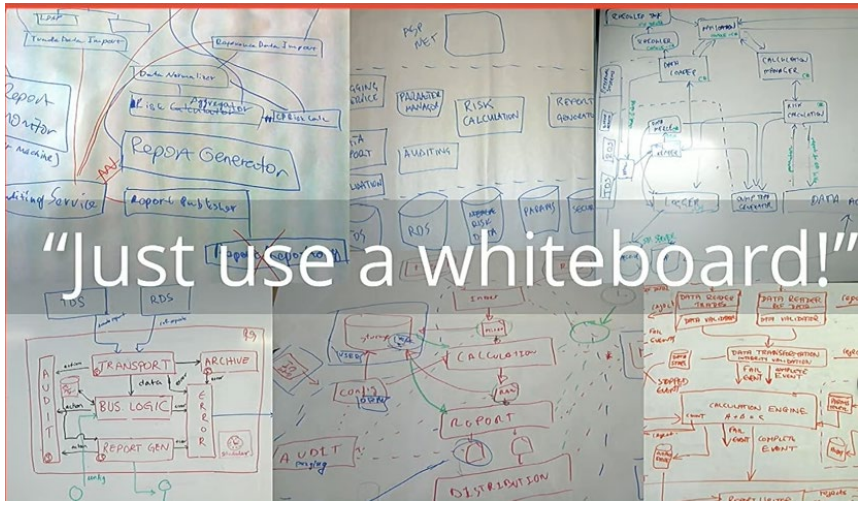
Try to model the behavior, the structure, logic, and infrastructure such that it satisfies all requirements and does not violate constraints. Architecture bridges requirements to implementation…

How to do that?

Use proper modelling diagrams for the different views and integrate them together.



Both extremes do not work in practice. What to do then?



Problems of Architecture Diagrams

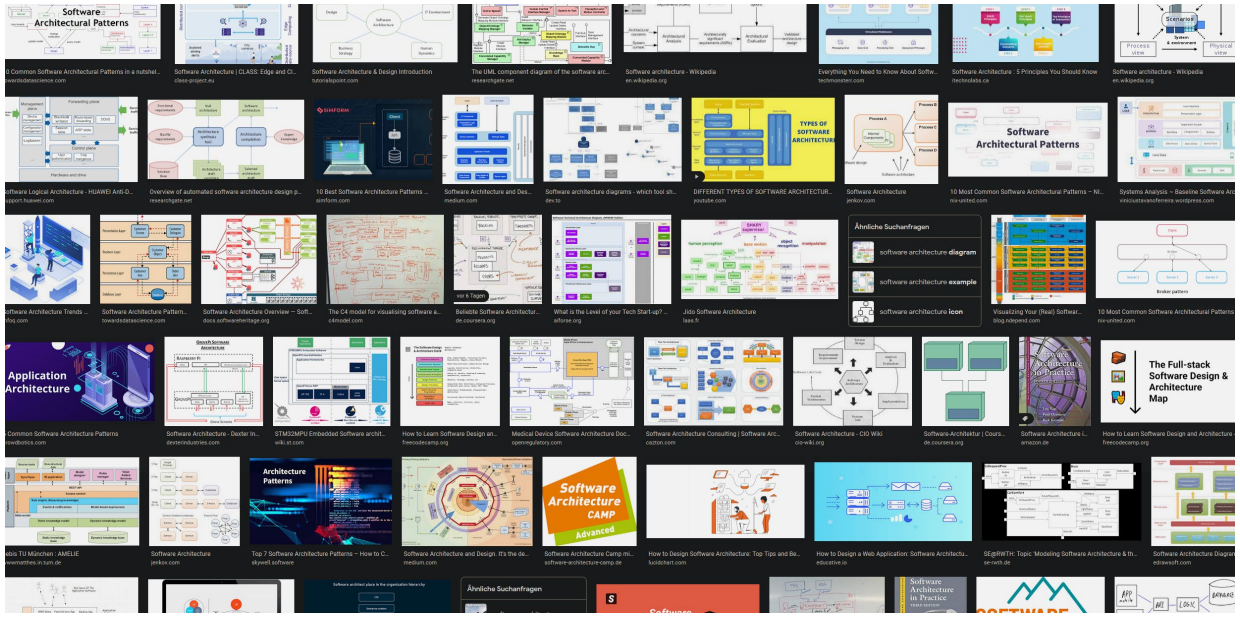
Boxes: Different shapes, sizes, colors, borders

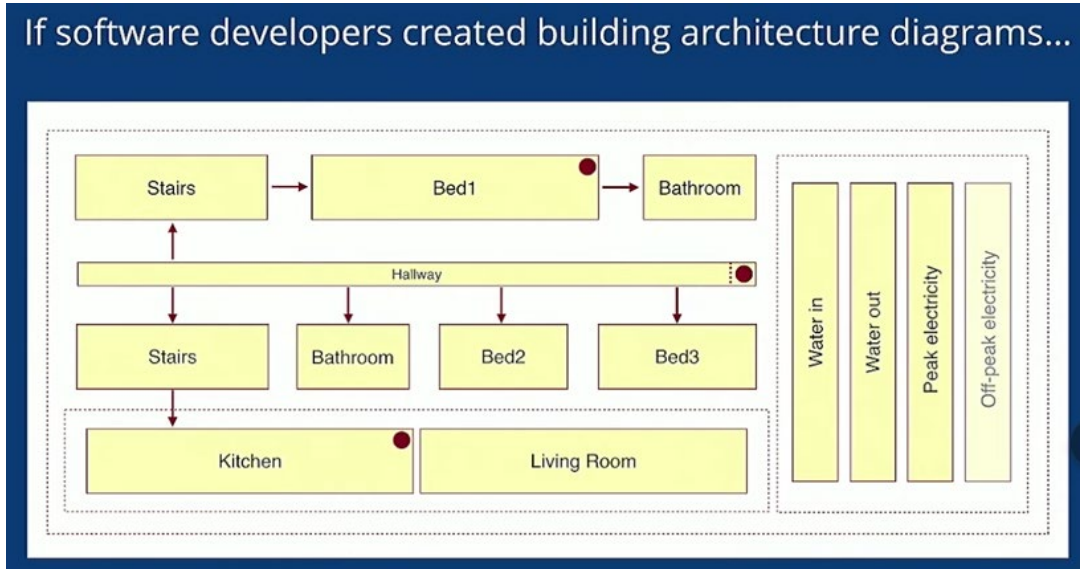
Lines: Shapes, arrows, colors, patterns, sizes, thickness, connections

Symbols and shapes: Different meanings, unclear pictures, acronyms

Missing elements: unconnected boxes, non-appearing actors, legend

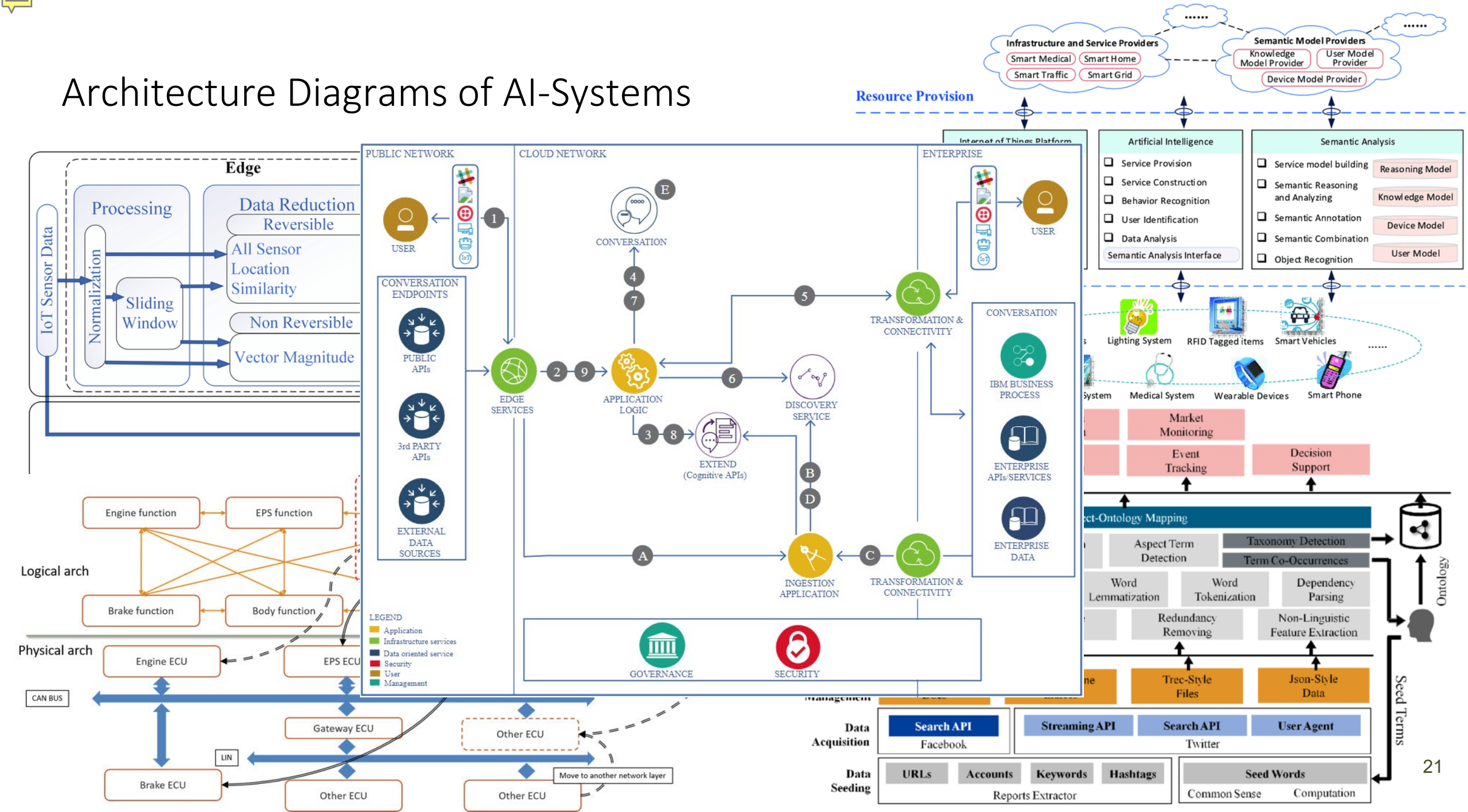
Arrangement: Nesting, layout





Source: Simon Brown

Architecture Diagrams of AI-Systems



Key Insight: Abstractions are Important; Notation second

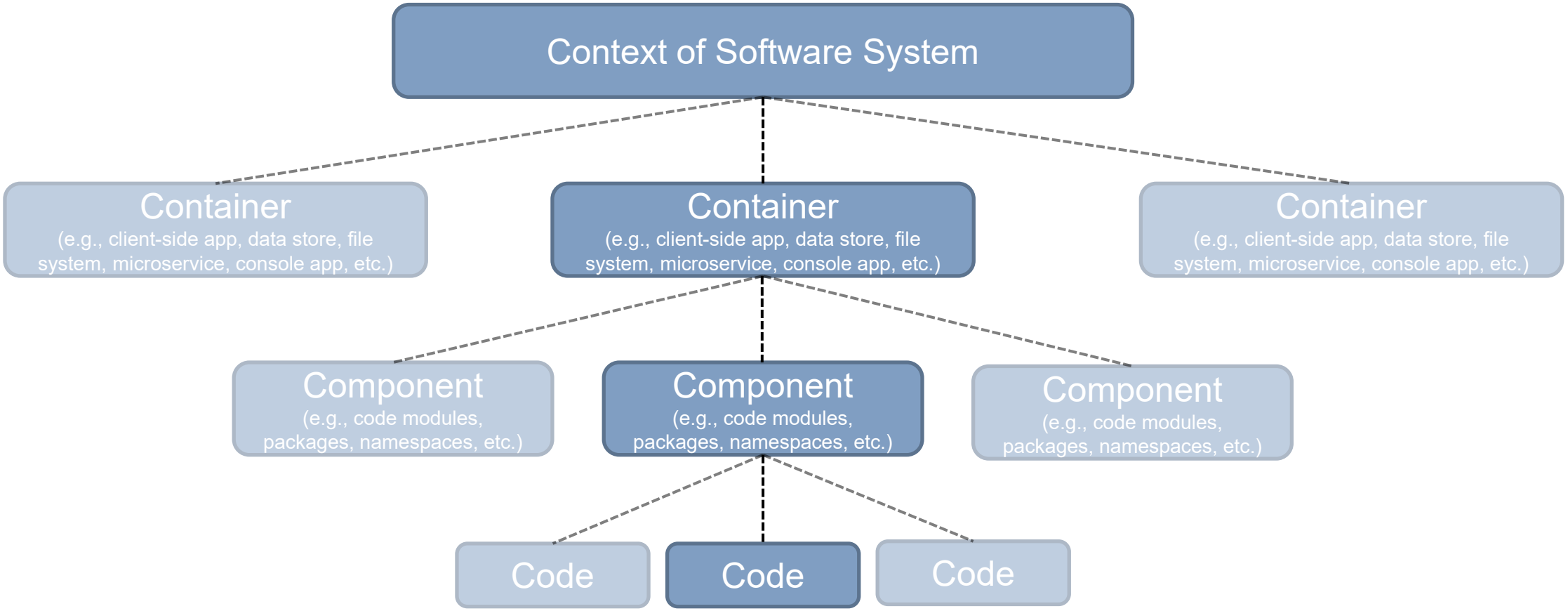
Notations are complex and could even mean different things to different people

Notations are often misused, misinterpreted, invented on the fly, or forgotten

Abstractions are the key elements of a system

No matter what notation, the abstractions are there

Modelling Architecture with C4

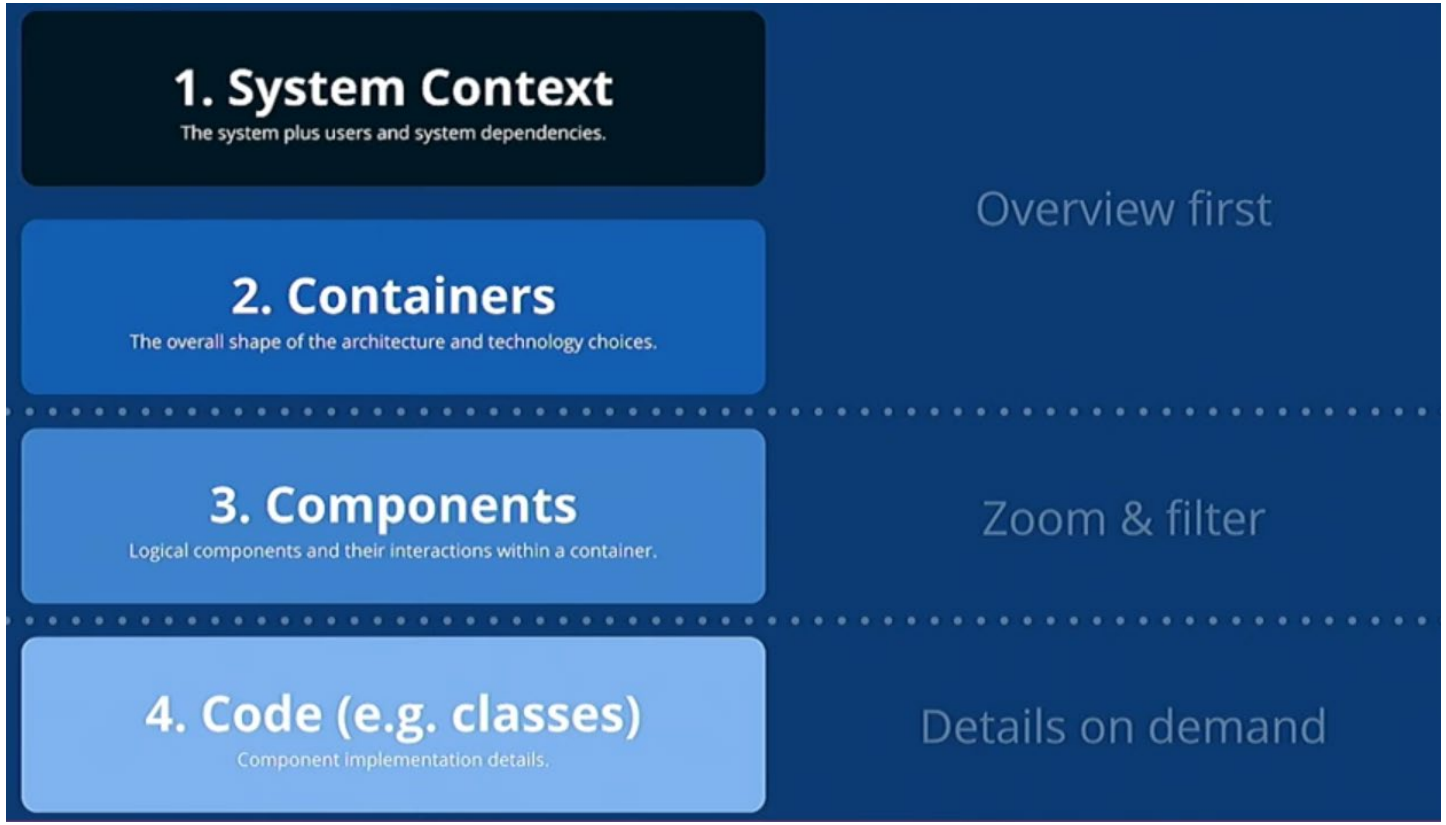


C4 Model Overview

Idea: Describe software system in different levels of detail.

There is no direction enforced (button-up or top-down).
It is not required to have all levels of detail.

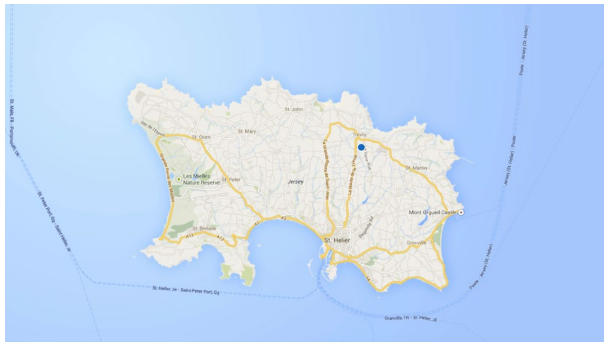
C4 = **C**ontext, **C**ontainers, **C**omponents, **C**ode



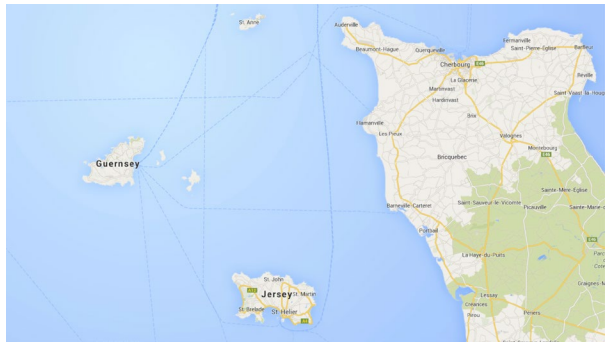
Diagrams == Maps for Developers

Huge detail; real-world look, but where am I?

Zooming out; we are on an island, but which and where?



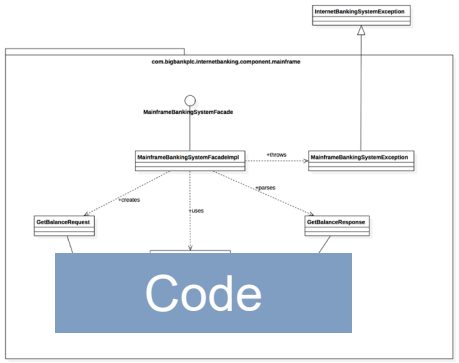
We are on Jersey, next to a larger island/continent?

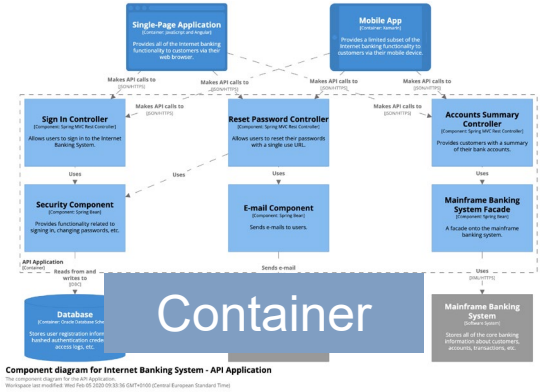


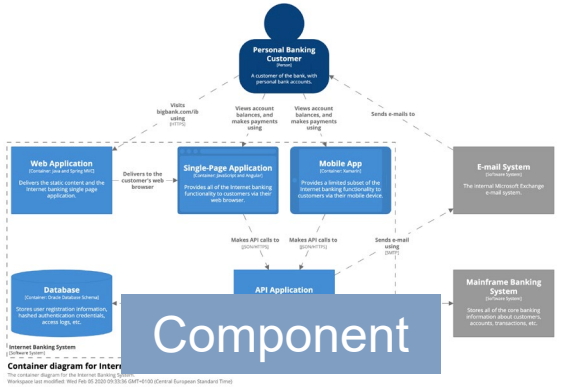
We are close to France!

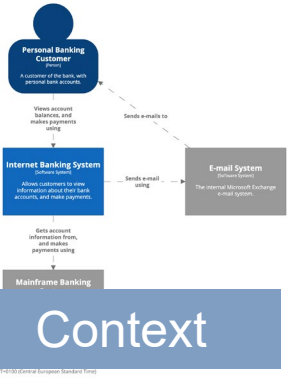
Zoom levels can tell different stories, highlight different aspects to different audiences!

In SW development, we have different audiences (users, customers, architects, testers, data scientists, etc.), so we need to tell and highlight different stories!

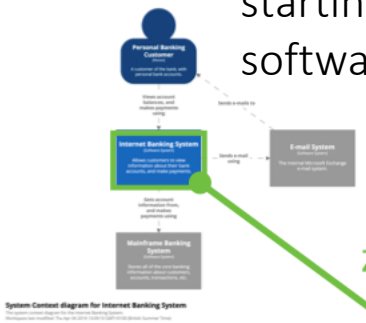






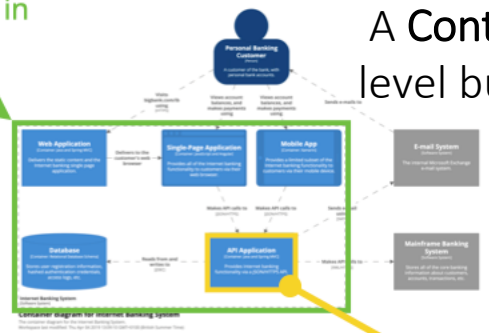


A **System Context** diagram provides a starting point, showing how the software system fits into the world.



Zoom in

A **Container** diagram shows the high-level building blocks of the system.



Zoom in

A **Component** diagram shows the components/modules of a container.

A **code** (e.g. UML class) diagram zooms into an individual component, showing how that component is implemented.

Zoom in

Level 1	Level 2	Level 3	Level 4
Context	Containers	Components	Code

Abstractions: Essential Elements

Person: Users of the system (actors, roles, etc.)

Software system: Software that delivers value to its users, including non-human users, such as other systems upon the modelled system depends on and vice versa.

Container: Application or data store that must run such that the overall system can work. Deployable/runnable unit or runtime environment.

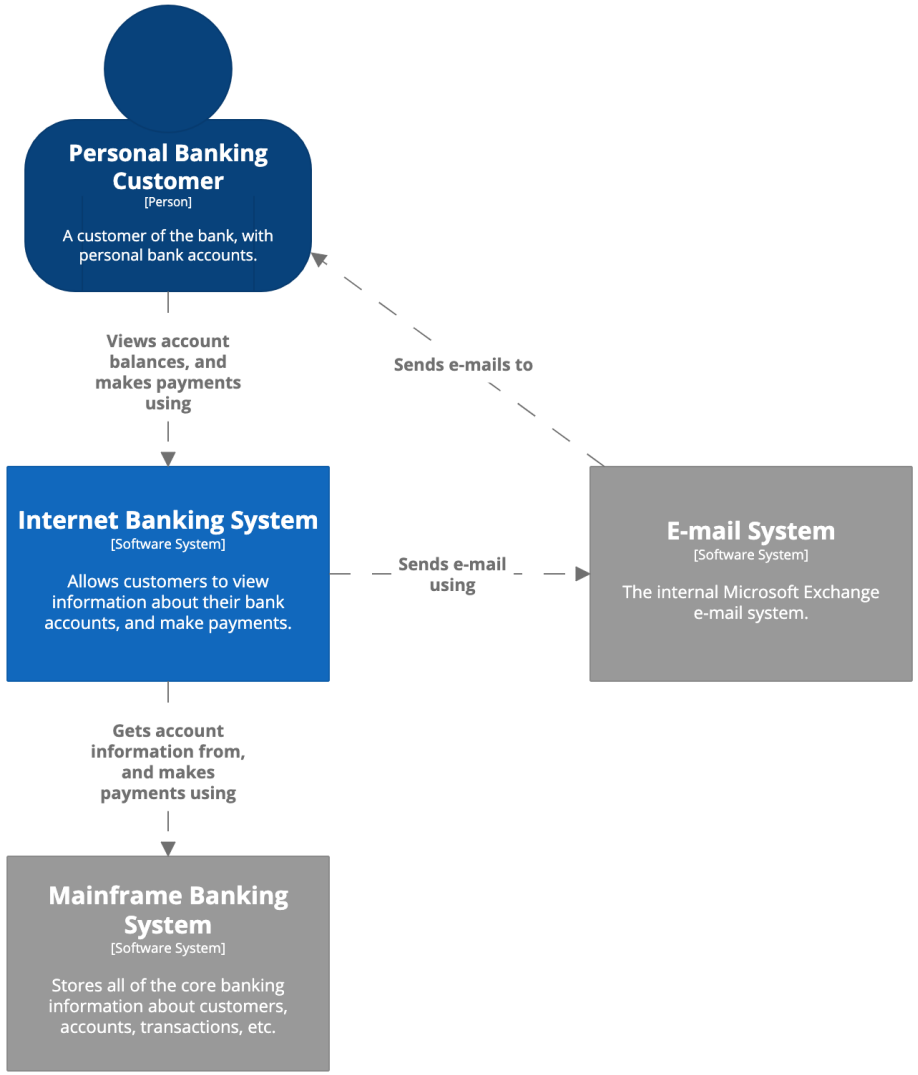
Component: Group of related functionality behind a well-defined interface. Components behind the same contain execute in the same process space.

C1: Context Diagram

Big picture of how your software system fits into the current landscape, including possible users and other systems.

Ideal for placing a system within other data consuming or producing systems and how they interact.

No technologies, no protocols, and other low-level details required. Intended for non-technical people.

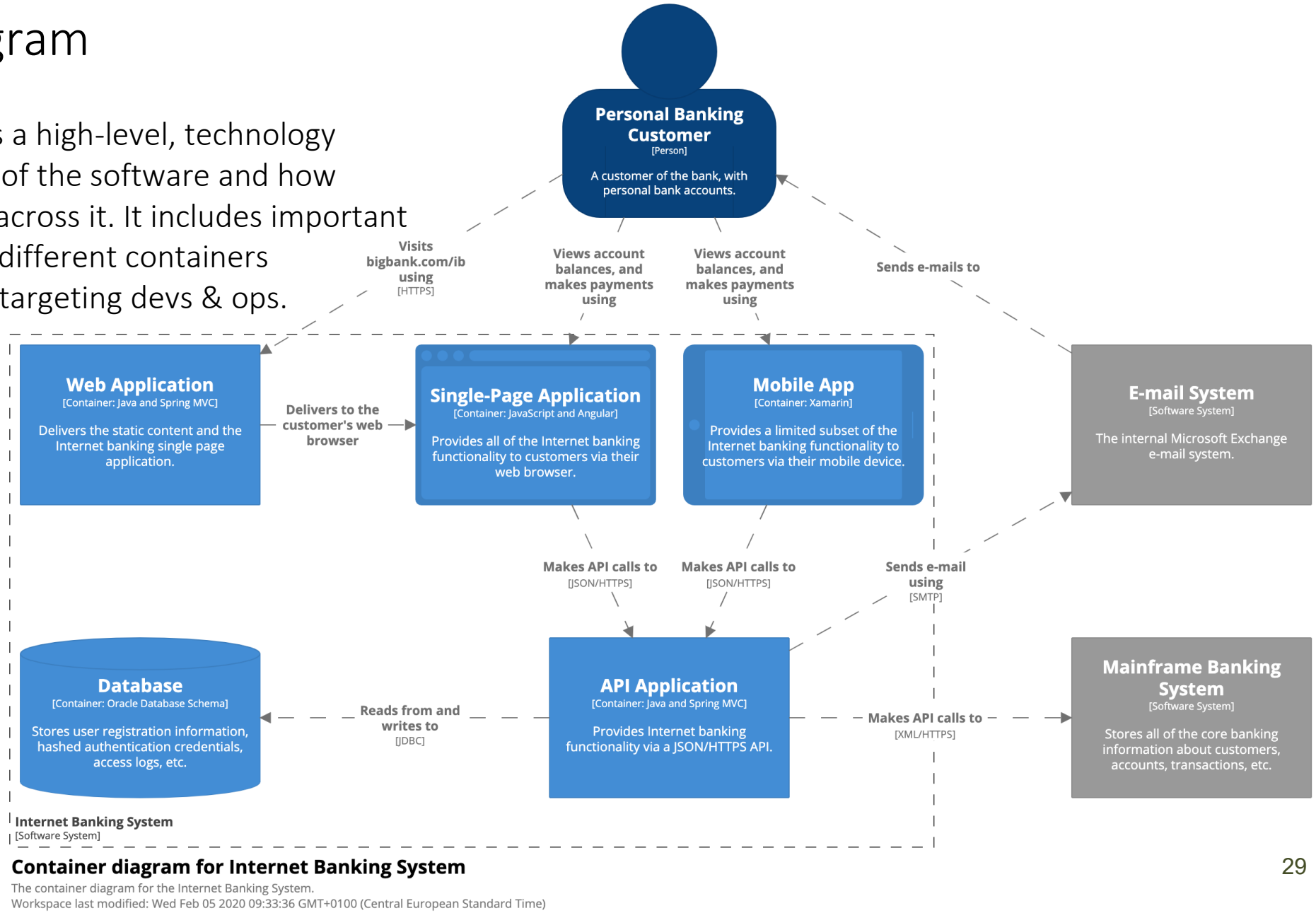


System Context diagram for Internet Banking System

The system context diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

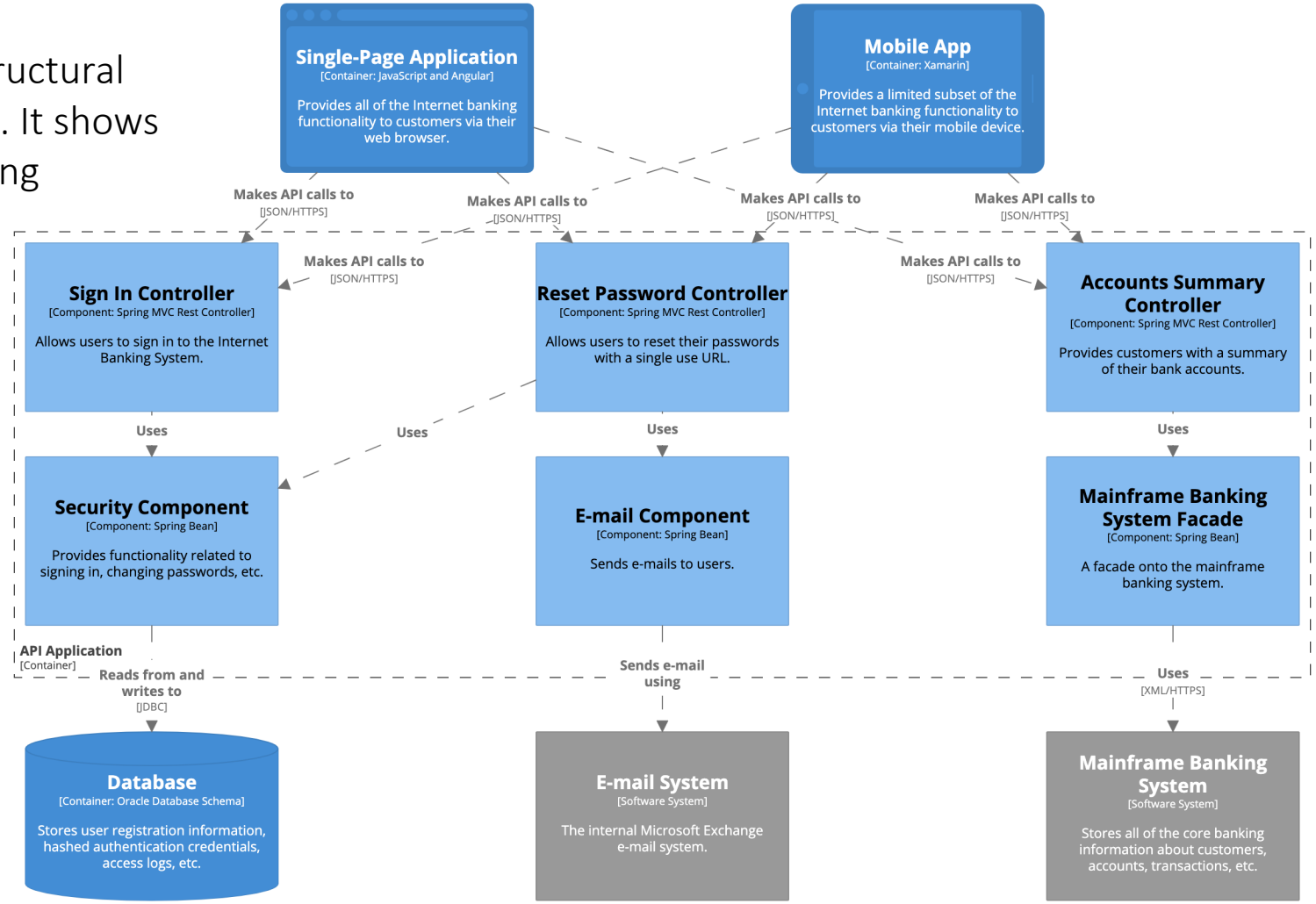
C2: Container Diagram

A container diagram represents a high-level, technology focused architectural overview of the software and how responsibilities are distributed across it. It includes important technological choices and how different containers communicate with each other, targeting devs & ops.



C3: Component Diagram

The Component diagram visualizes the major structural building blocks (i.e., components) of a container. It shows their responsibilities and the technology, including implementation details.



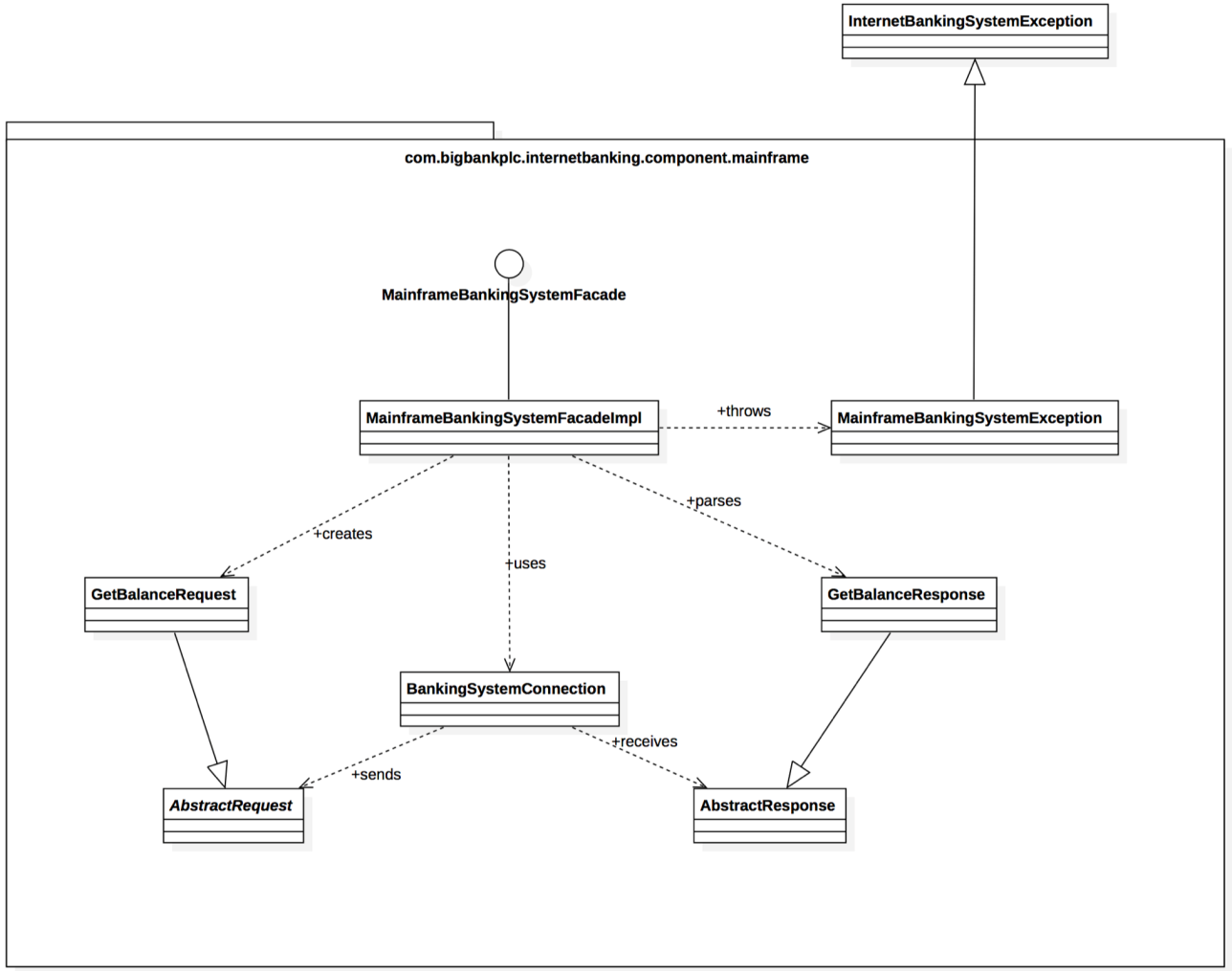
Component diagram for Internet Banking System - API Application

The component diagram for the API Application.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

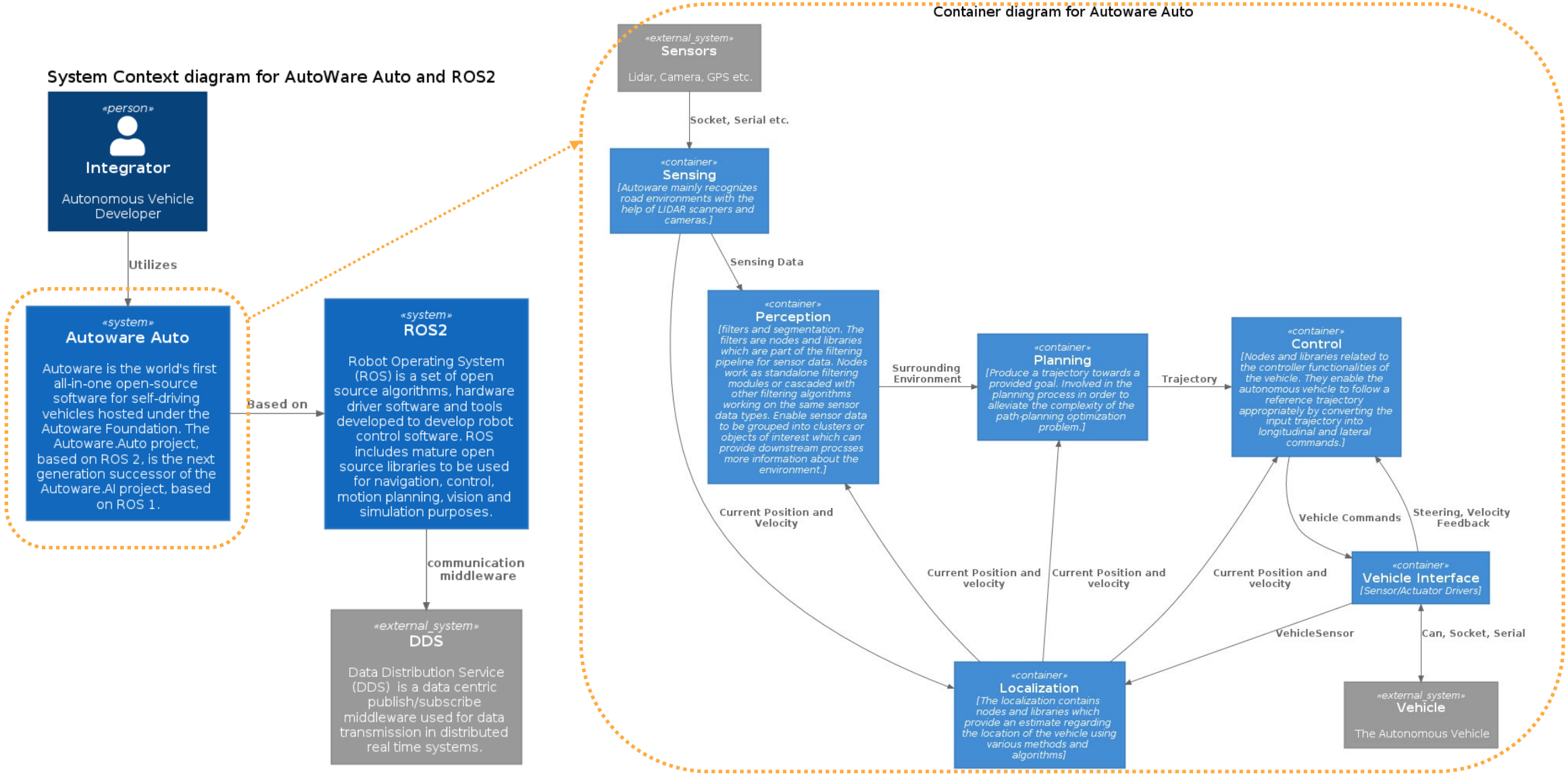
C4: Code Diagram (optional)

The code diagram is not defined and depends on the use case. Often used is some kind of UML class diagrams, entity relationship diagrams, or state or sequence diagrams.

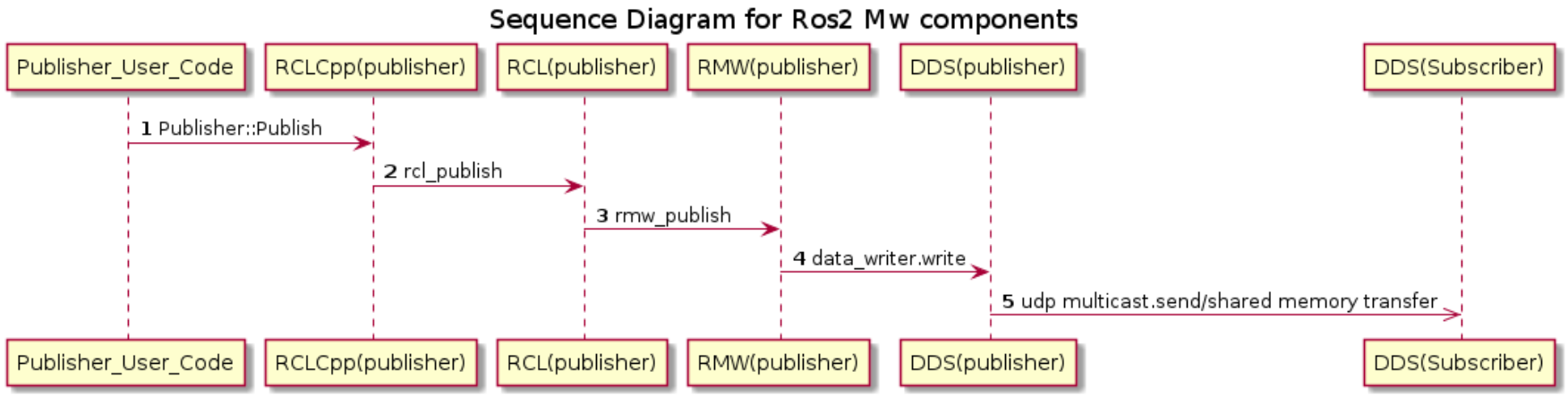
This diagram may be generated by the IDE. Only recommended for very complex components.

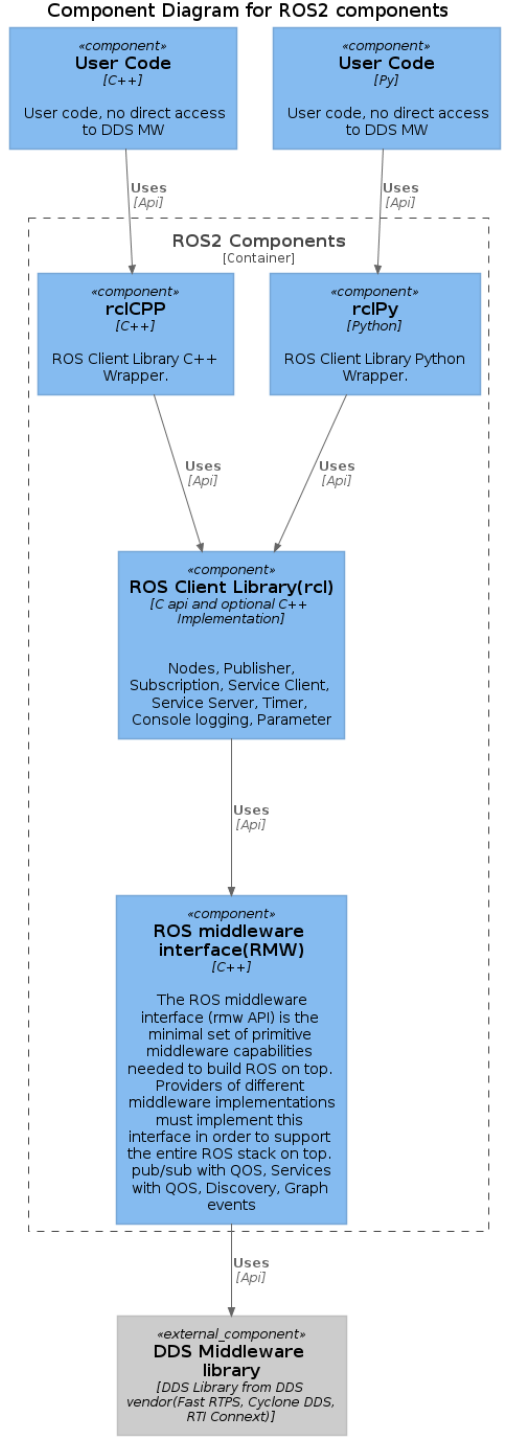


Example: Autonomous Robotic with C4

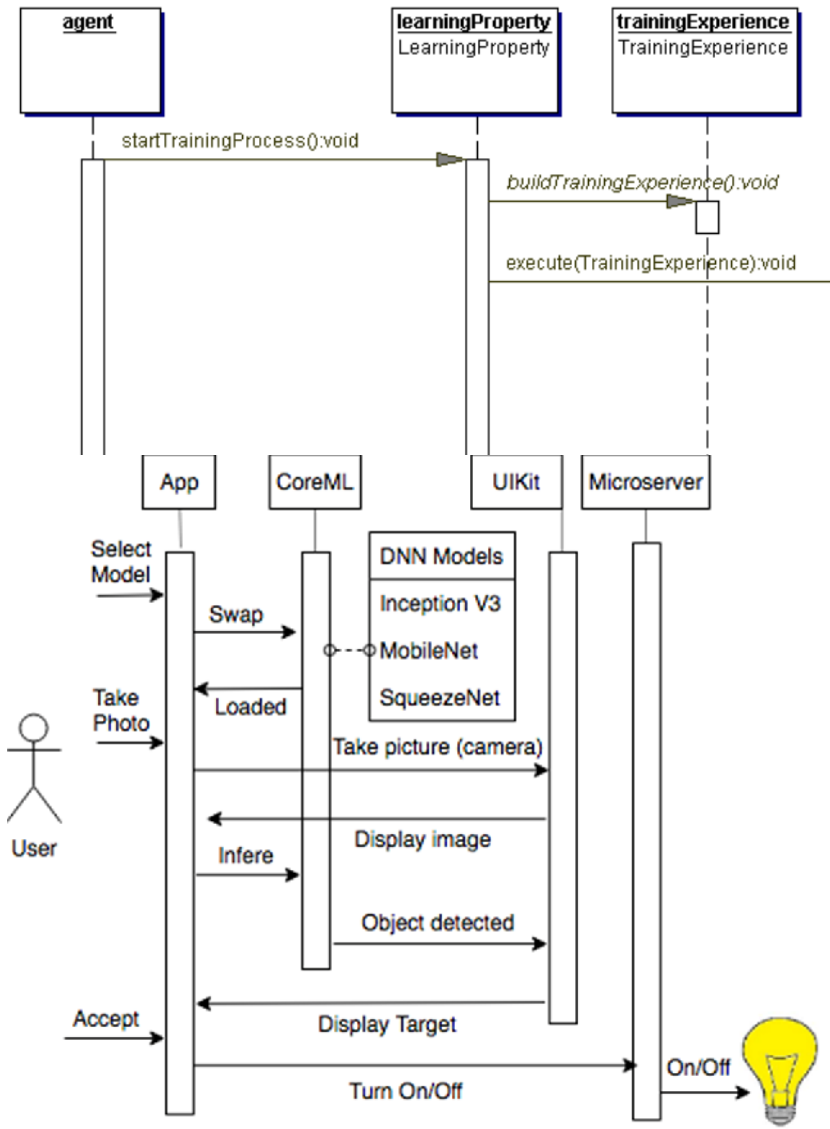


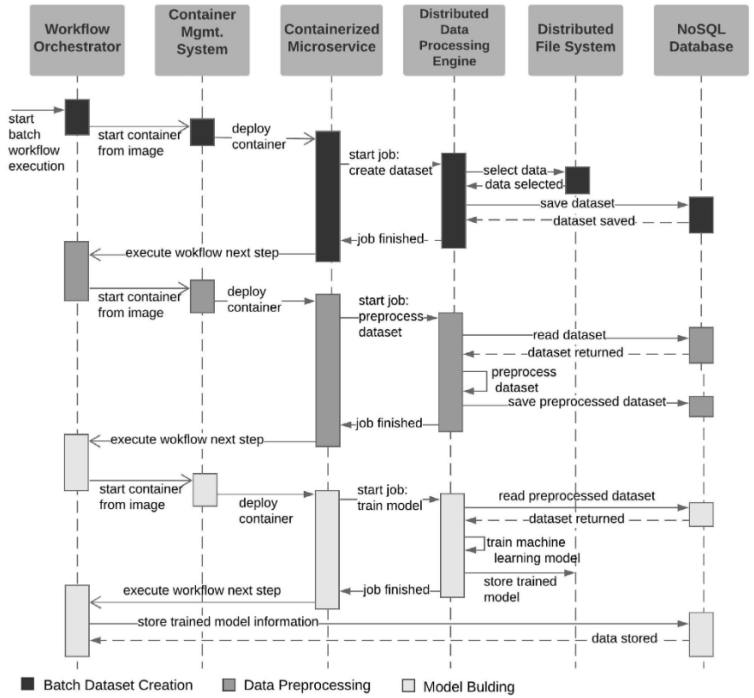
Example: Autonomous Robotic with C4

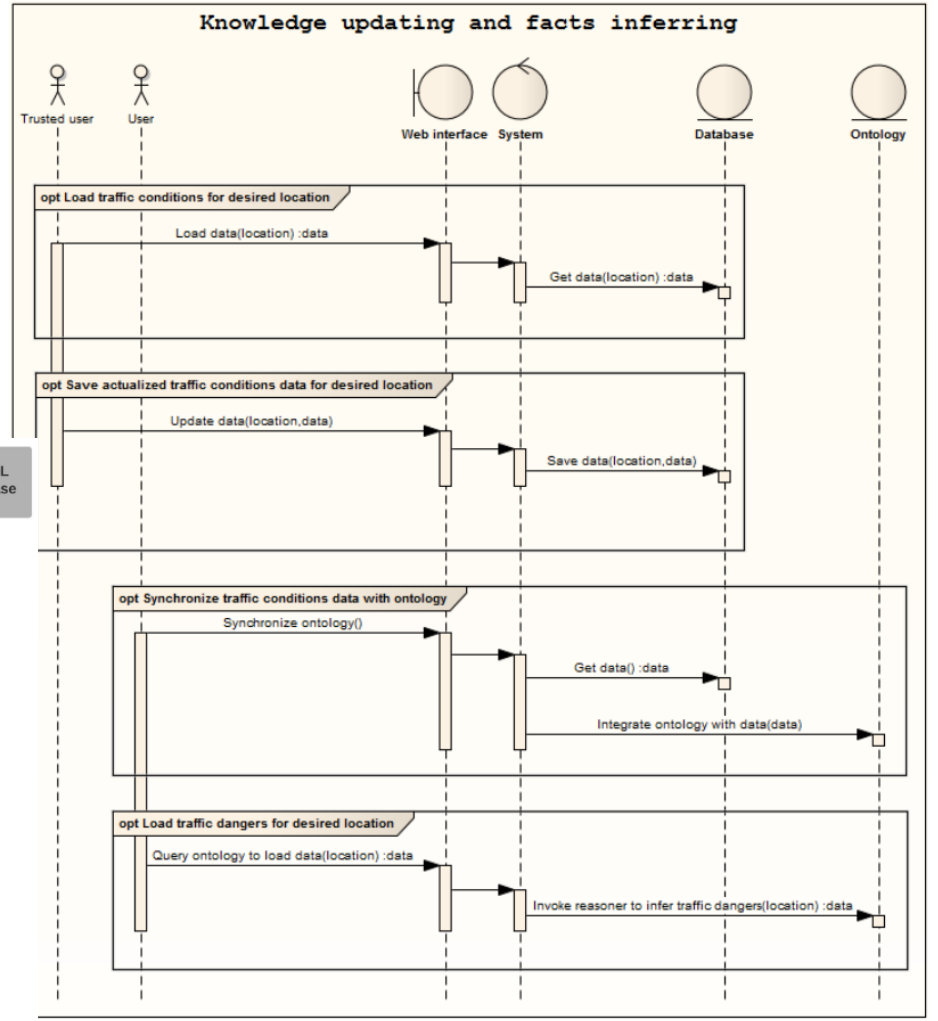




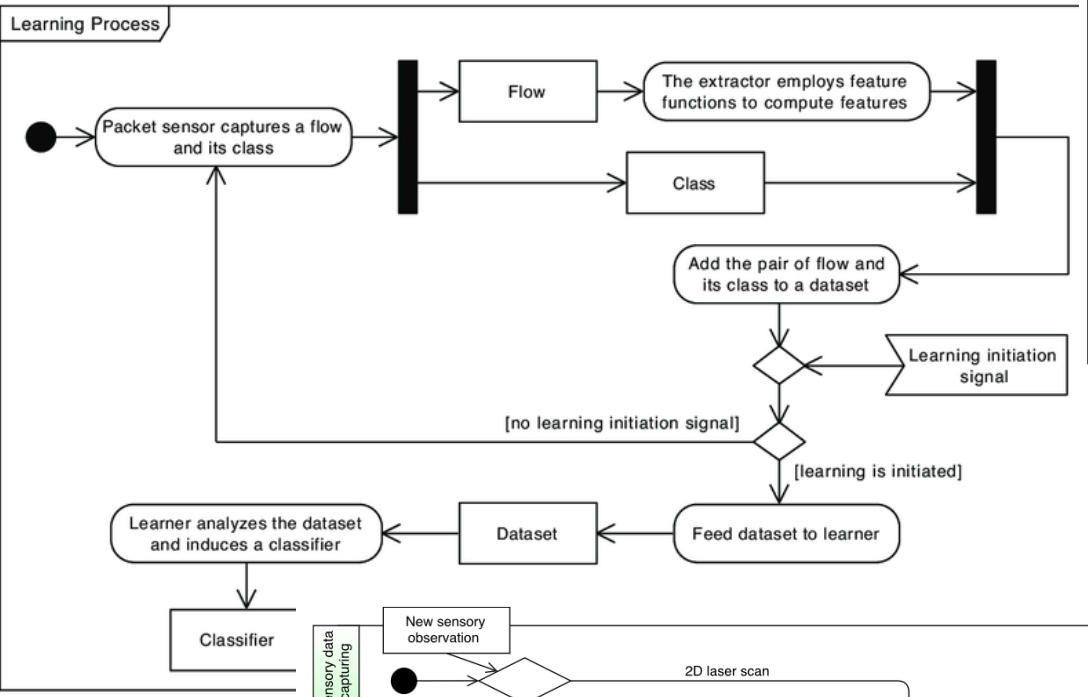
Modelling: Sequence Diagram (UML)

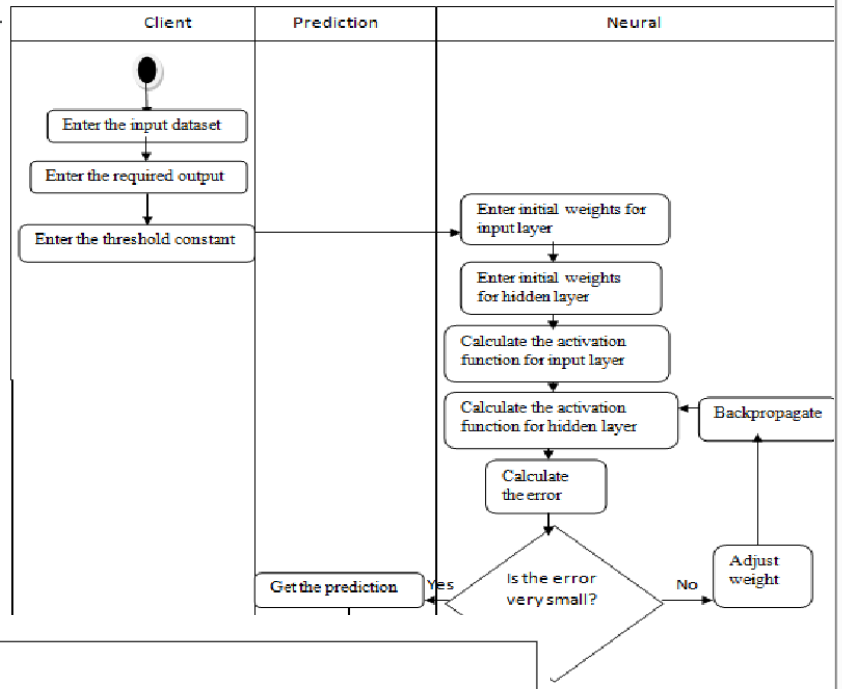


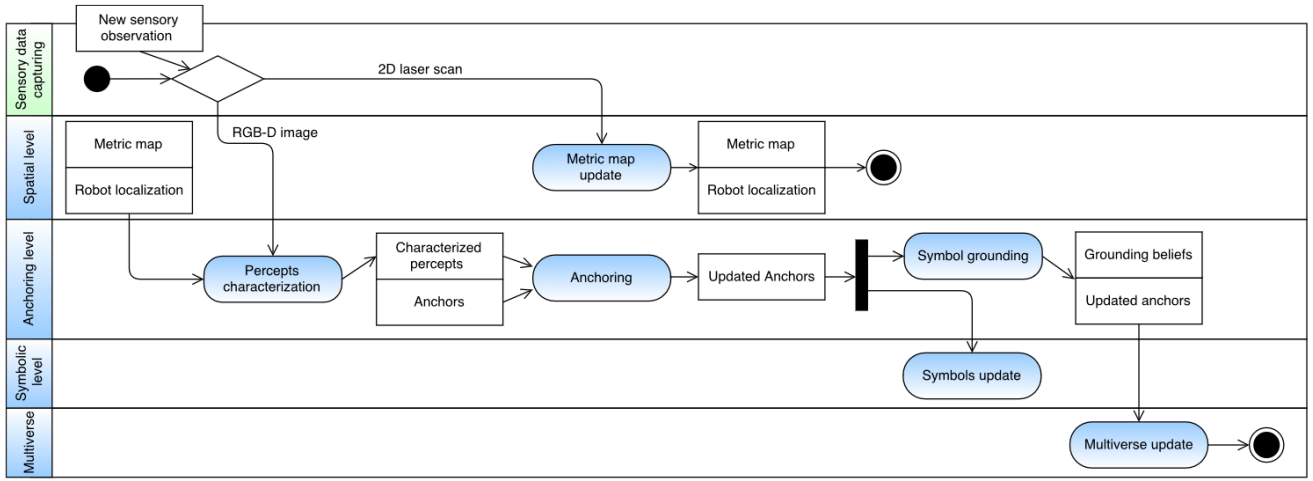


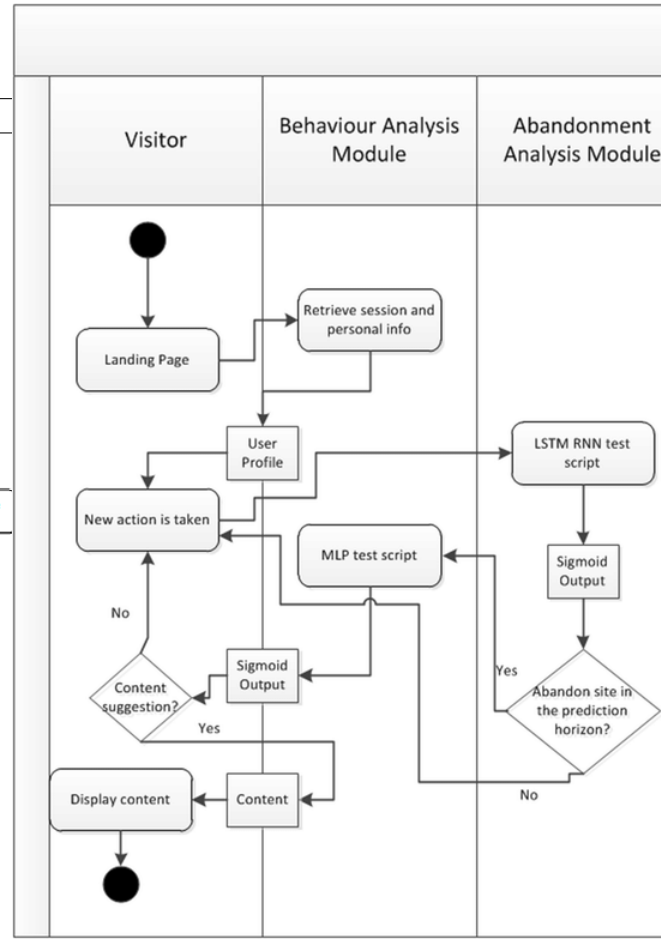


Modelling: Activity Diagram (UML)

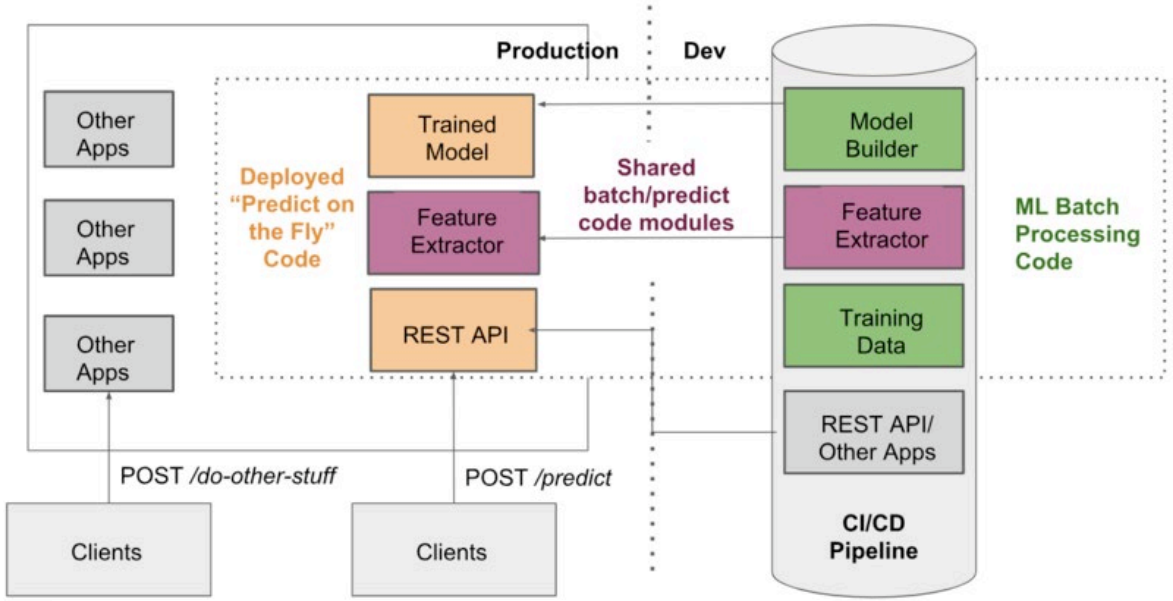


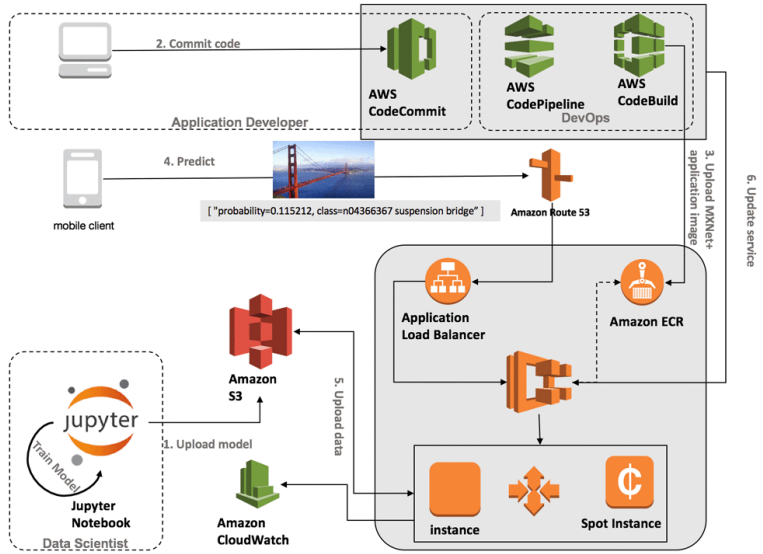


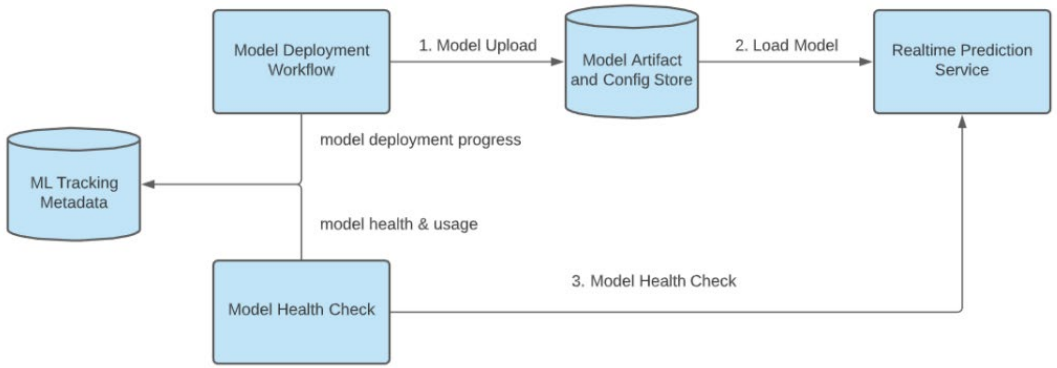


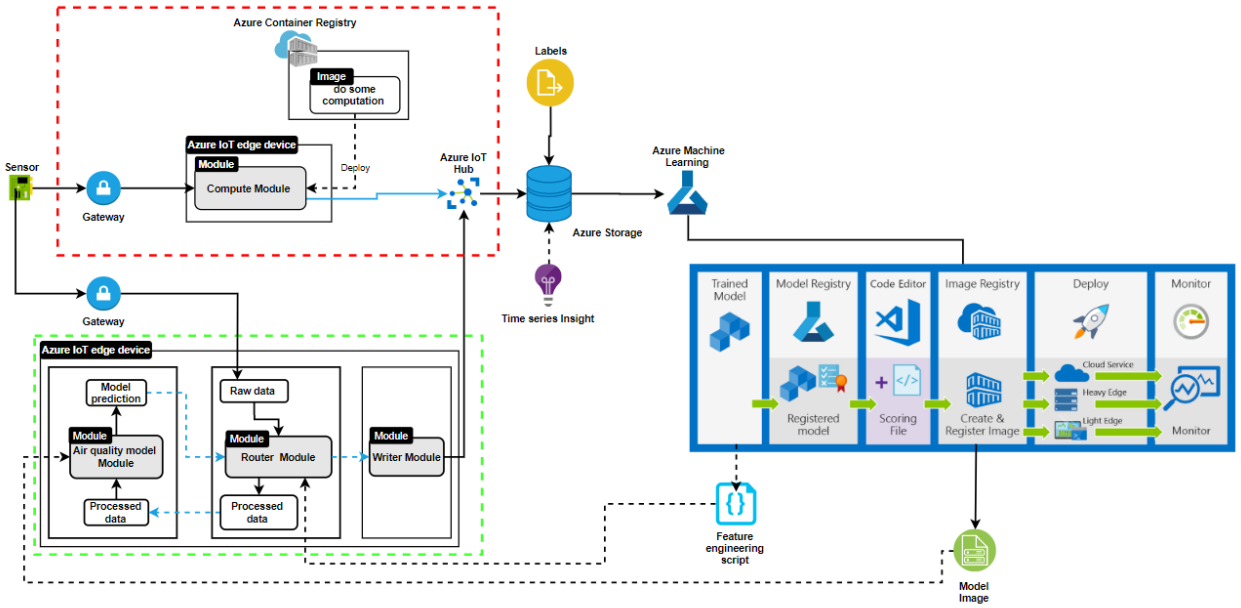


Modelling: Deployment Diagram









Good Architecture == Team Effort

Avoid insular knowledge and get multiple perspectives (SW, ML, customer, etc.) by having a team decide on the architecture

Skilled professionals know the tools and technologies to decide on

Domain experts know interfaces and consequences to environment and use case

Managers know organizational constraints (committees, politics, resources)

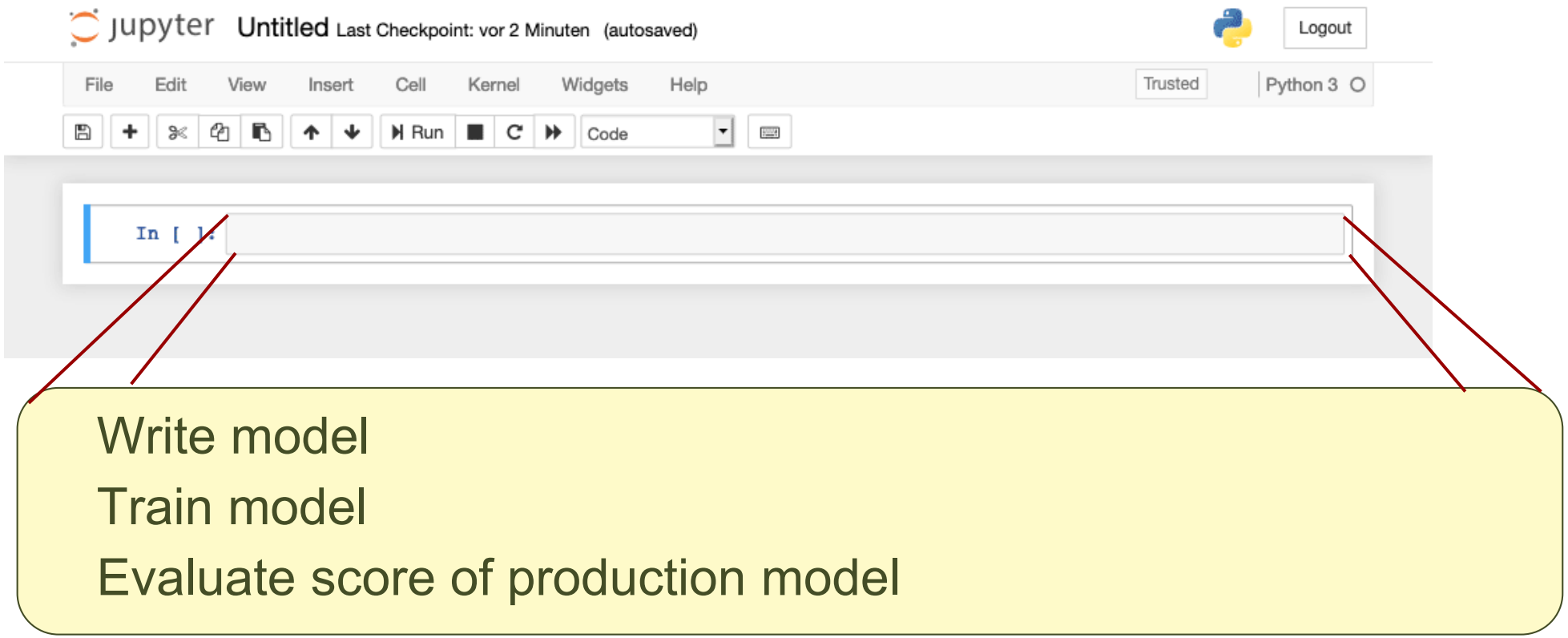
Software Quality

Principles from software engineering for software quality apply to AI-enabled systems

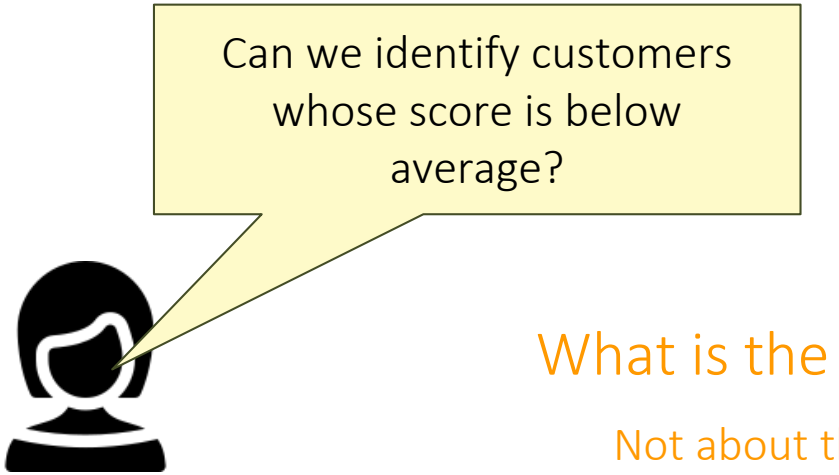
- SOLID principle / GRASP pattern / etc.
- Design patterns (see specific ML design patterns later)
- Documentation (e.g., comments, architecture description records, etc.)
- Automation (DevOps becomes MLOps)
- Testing (unit, integration, acceptance + ethics, data, model testing)
- Processes (align ML heartbeat to SW heartbeat, team composition, etc.)

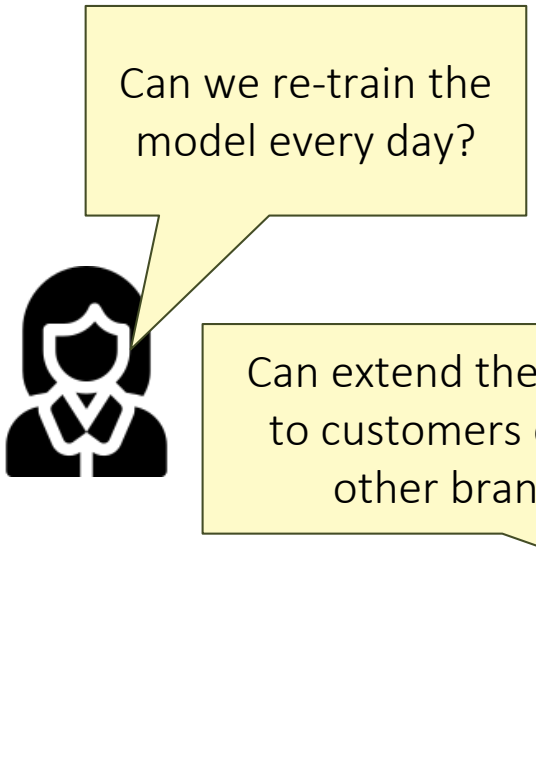
Topic II:
Architecture Components of ML-Enabled Systems

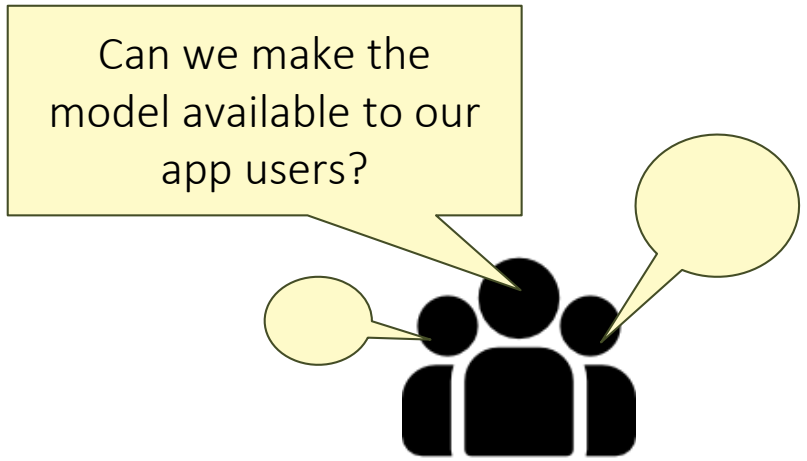
Typical Project Start



Project Continues…







What is the nature of these questions?

Not about the business case, not about the ML technique, not about how we train, but all about how we productionize a model!

Architectural Design Decisions

Which component to build and how to integrate it with

- Other system components?
- Communication and interactions?
- Data store?
- Runtime environment?
- Logging stack?
- Monitoring capabilities?
- 3rd party solutions?

How to decide on

- Exchange protocol and format?
- Deployment tool?
- Scaling solutions?
- …

Where to put the intelligence?

- At the user?
- At a data center?
- At an intermediate place?
- In the software system?
- At multiple places?

Systematic Architectural Decisions

System landscape

Decide on …

Tools und runtime environment

Use all-in-one cloud providers or integrated (local) solutions

Intelligence location

Decide on …

Where the model should live (bringing runtime, context, and model together)

How to deliver predictions to the use case

System architecture

Decide on …

System components and how they are connected

The degree of automation and scalability

System Landscape: Integrative

Idea: Review system and tool landscape of your company or the company in which the ML module is deployed or integrated to

Rational: ML module will fit into the software system landscape and can use existing knowledge and infrastructure for deployment

Example: Order module

Module architecture: Microservice

Language/framework: Java / Spring Boot

Storage: Postgres

ML Module

Language/framework: Python/Flask

Storage: AWS RDS

API protocol / format: REST / JSON

Runtime environment: Kubernetes

CI tool: Github Actions

Logging: Elasticsearch

Monitoring: Prometheus

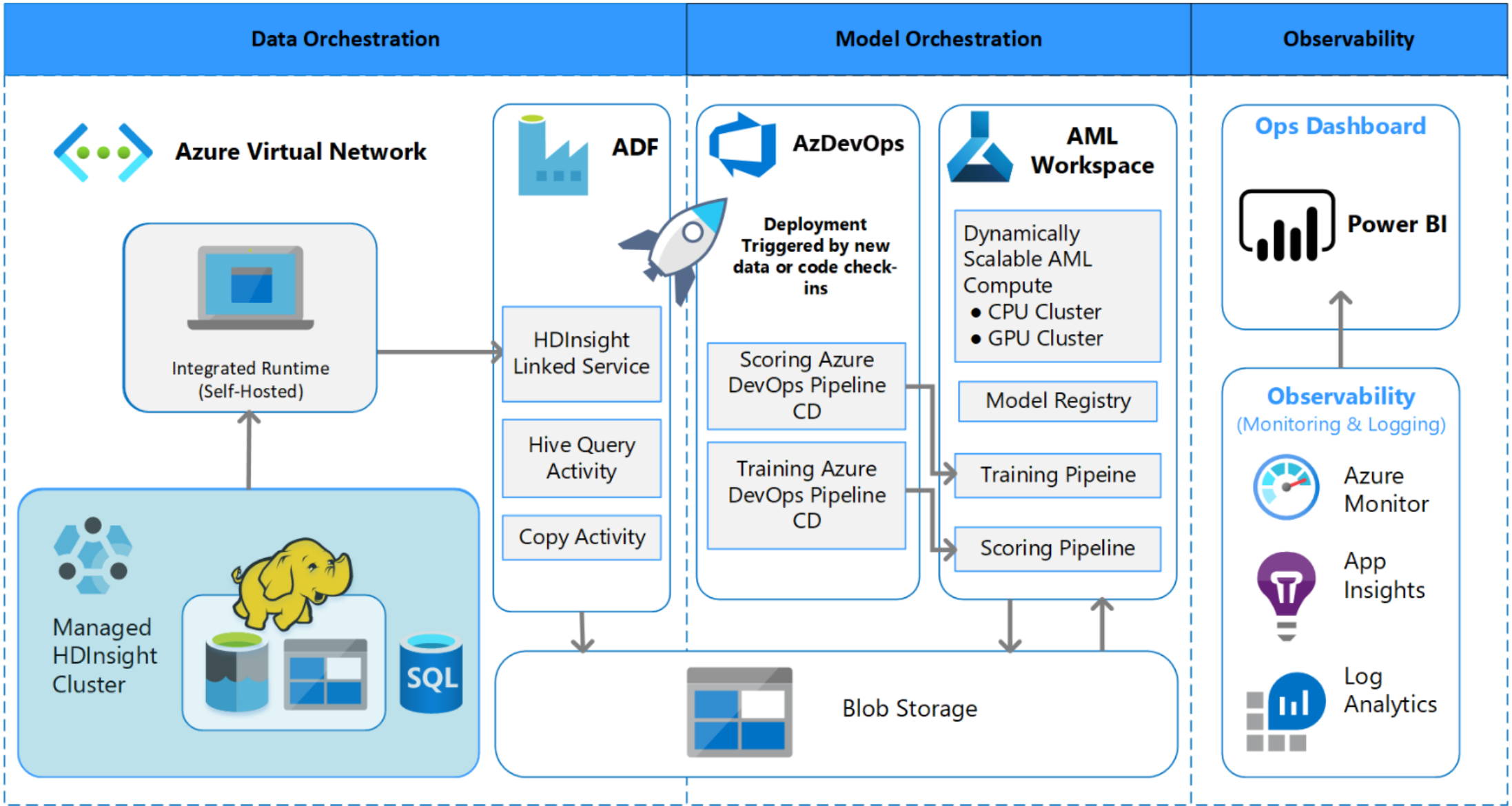
System Landscape: Cloud Ecosystem

Idea: Use one of the existing ecosystems (e.g., MS Azure, AWS Sagemaker, etc.)

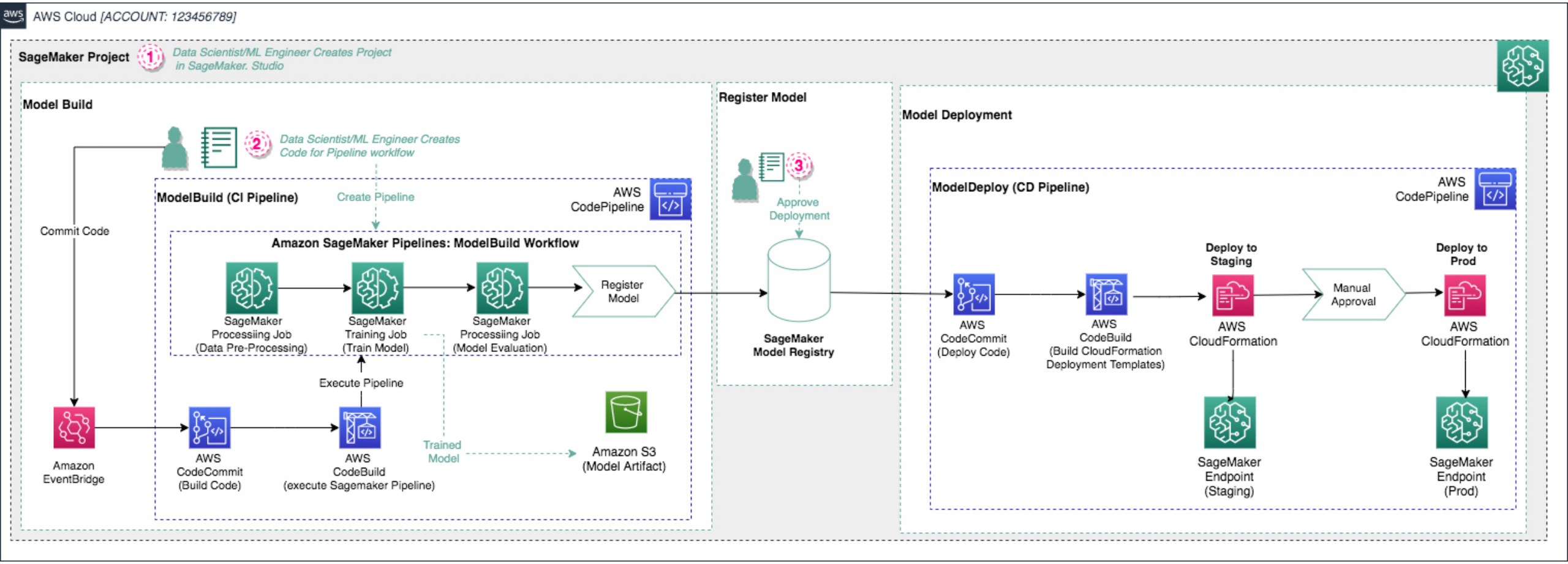
Rational: Use an already integrated solution with a powerful feature set and infrastructure support. Avoid self-implementations and integration (boilerplate code) of different tools and frameworks. Use existing scalable hardware and infrastructure with no administration cost.

Pitfalls: Vendor lock-in, that is, hard to move to another platform. Can be costly. May be in conflict with regulations (where is my data stored?). Better tools for some aspects from other vendors. Some integrations with own modules may be difficult.

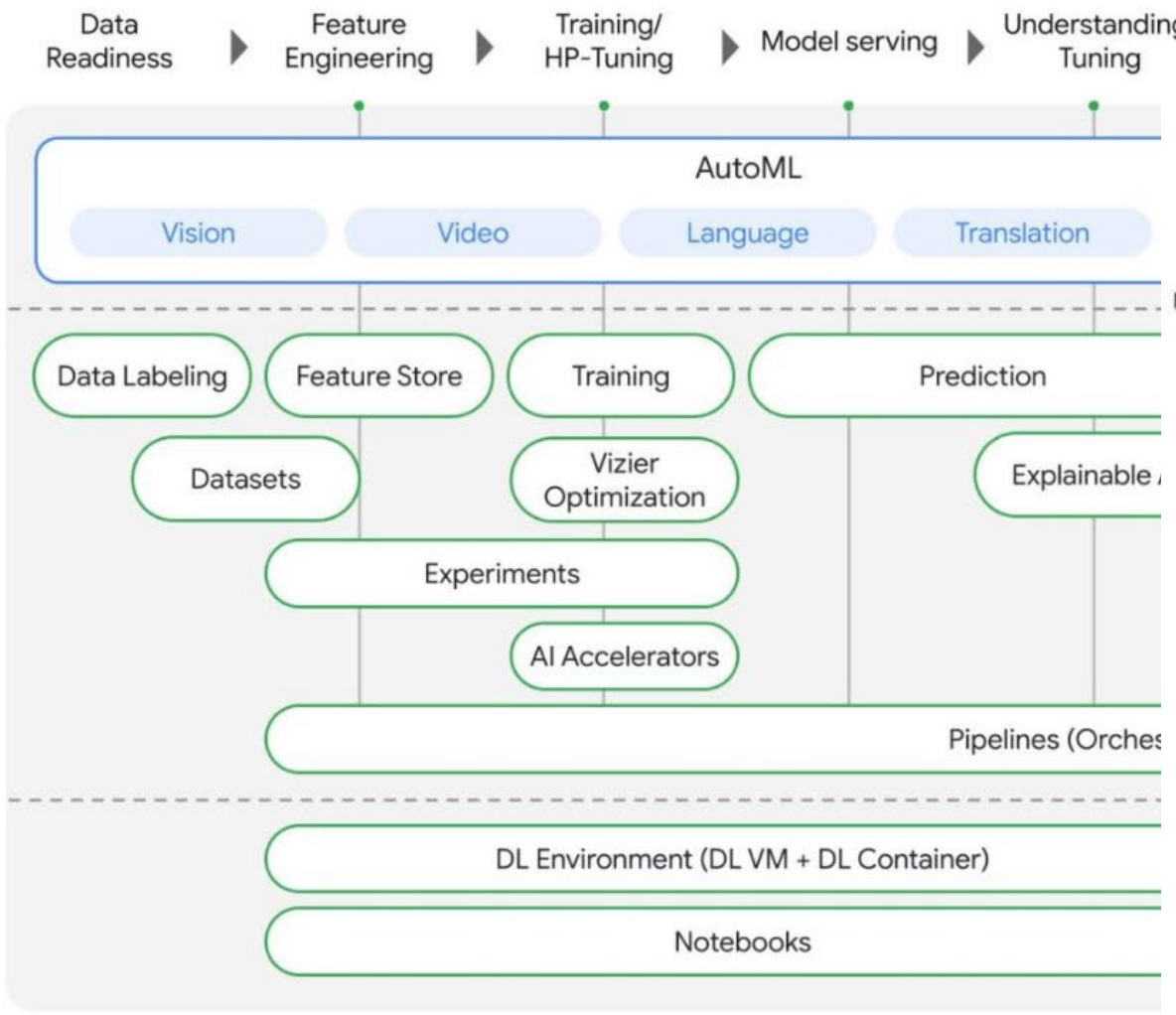
Microsoft Azure AI

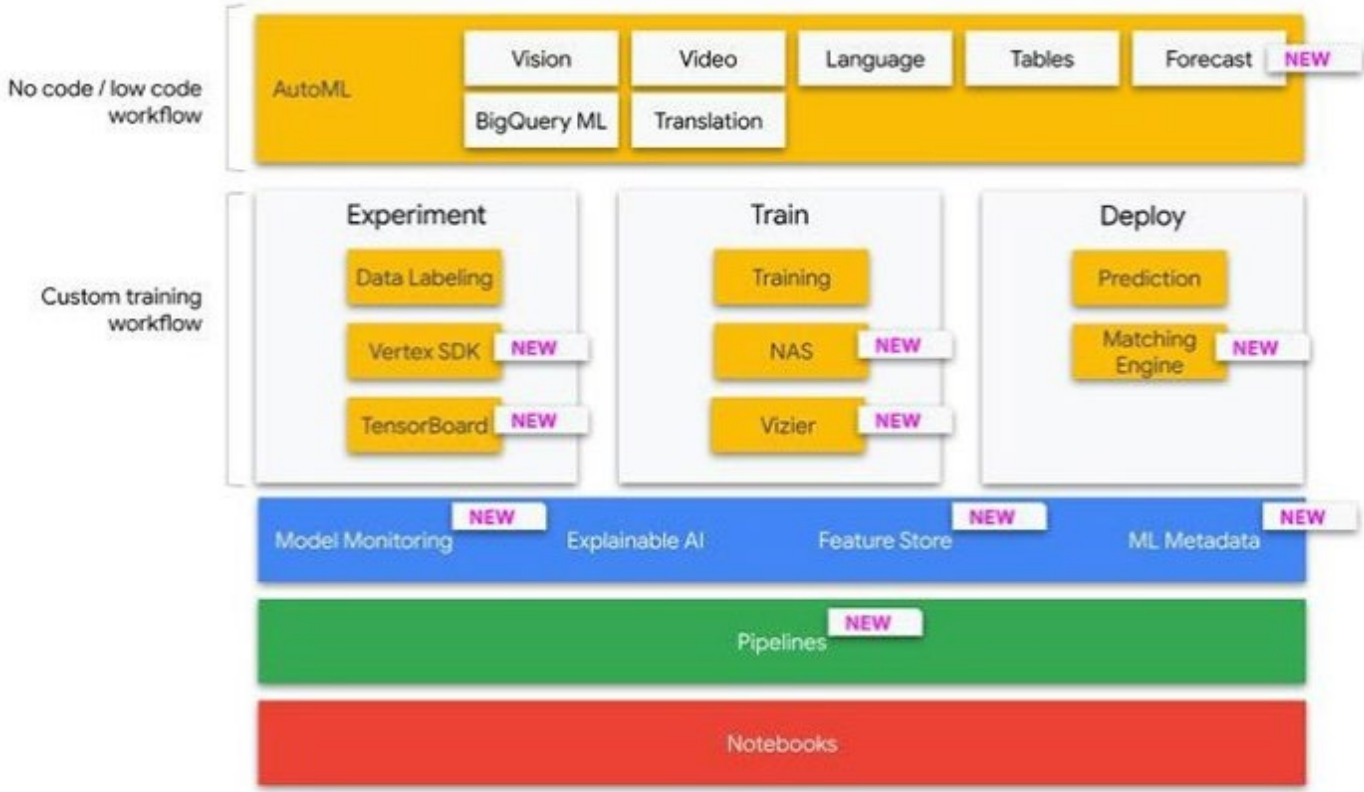


Amazon Sagemaker



Google Vertex AI





Pros+Cons: Cloud / ML Platform Provider

+ Standardized, “plug&play” pipelines for data preparation, training, and deployment
+ Easy to close a feedback loop
+ Easy to scale for massive amounts of data

- Limited to few algorithms (PyTorch vs. TensorFlow vs. MS Azure AI)
- Vendor lock (tied to a company’s infrastructure)
- Can be expensive
- Specialist knowledge is lost if working for another provider

Solution: Open source stack with, for example,

Systematic Architectural Decisions

System landscape

Decide on …

Tools und runtime environment

Use all-in-one cloud providers or integrated (local) solutions

Intelligence location

Decide on …

Where the model should live (bringing runtime, context, and model together)

How to deliver predictions to the use case

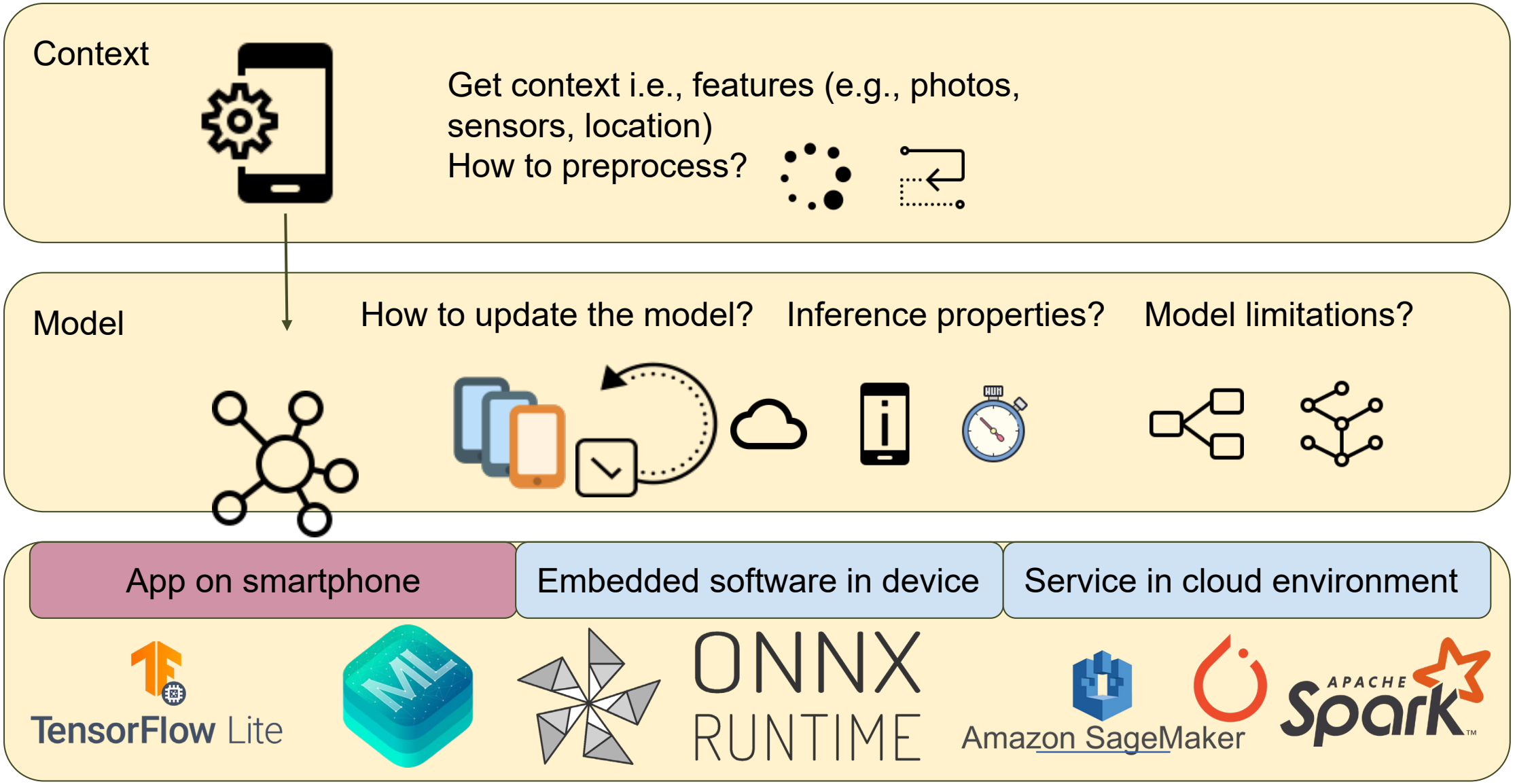
System architecture

Decide on …

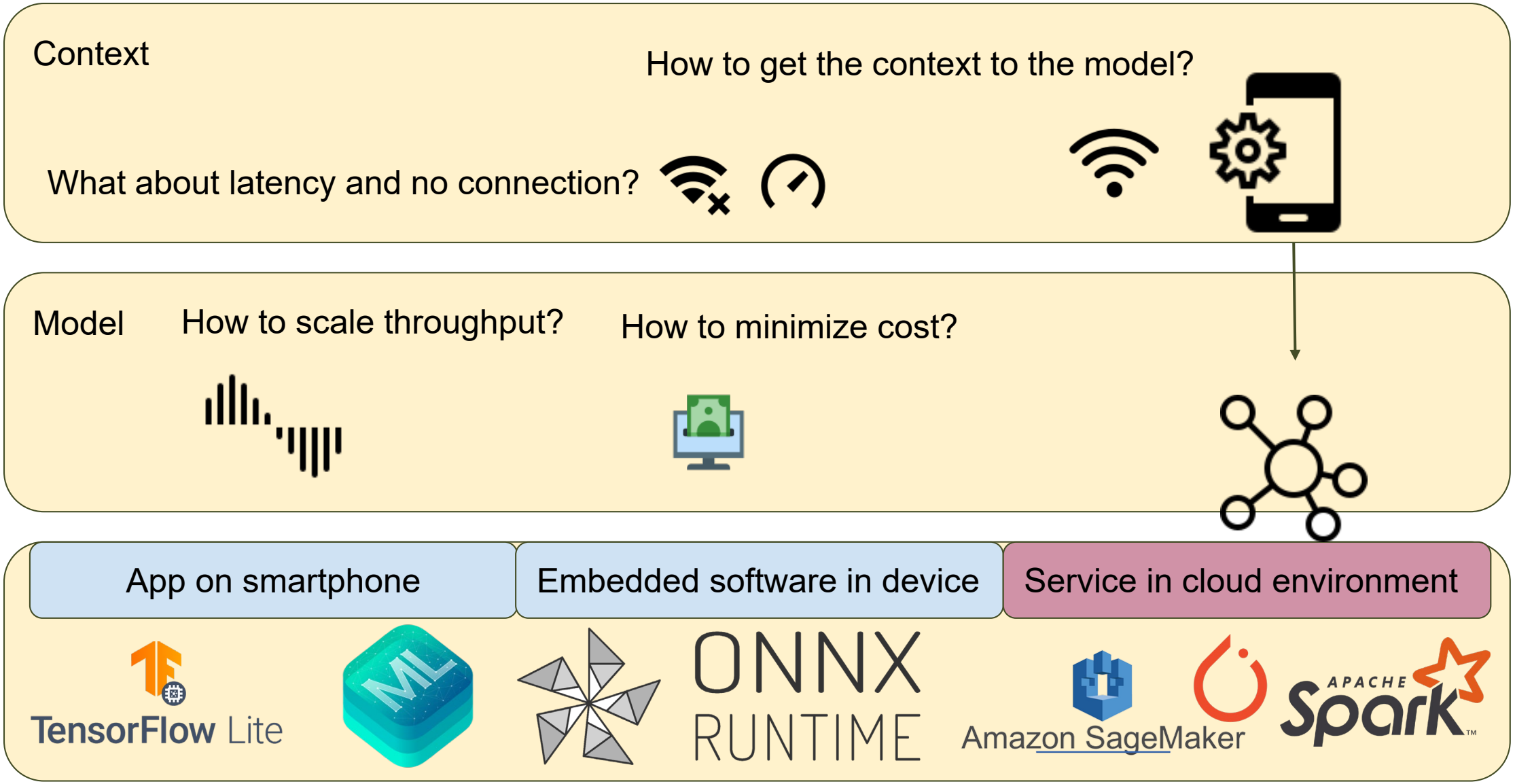
System components and how they are connected

The degree of automation and scalability

Intelligence Location: Aspects



Intelligence Location: Aspects



Latency Aspects: Model Update

Latency issues can occur when updating the model (cloud location vs. remote location with no stable internet connection).

Factors for consideration:

- Frequency of model updates (new data, new insights, problem changes)
- The need for an update (risk of costly mistakes)
- The availability of the runtime for an update
- The disruptiveness in experience of an update
- The complexity of the architecture to maintain different versions of a model (

Latency Aspects: Inference

To predict something, the system needs to gather the context and preprocess it to obtain features and send these features to the model, wait for its execution and send the result back. Depending on the application scenario each step introduces more or less latency and users may accept more or less latency.

Factors to consider:

- Synchronous or async communication with intelligence
- Number of steps involved and amount of processing effort
- Responsiveness of intelligence

Cost Aspects

Two factors drive the cost: the degree of distribution in the architecture and the execution of the intelligence.

Distribution cost:

- Bandwidth and communication cost for model updates (size of model, frequency of updates), monitoring data, feedback data
- Number of API calls / DB accesses etc. are cost factors for cloud environments

Execution cost:

- Bandwidth cost for sending the context / features to the model
- CPU cycles, RAM, GPU, TPU, energy costs for inferring the model
- Single vs. batch processing

Locations for the Model and Runtime

Static intelligence in the product (trained once, bundled and shipped with SW)

- Similar to SW heartbeat (updates ship with SW updates)
- Might be good enough for many use cases, but not for open-ended, time-changing, or hard problems

Client-side intelligence lives and executes on the client and obtains model updates from time to time

- Updates include new model, new preprocessing code, new thresholds for API function
- Unclear when to update and how to operate multiple versions of a model or how to secure it

Server-centric intelligence runs a real-time service

- Easy to update a model and detect shifts in data that are send to the server
- Requires heavy scaling of the infrastructure and cost of inference is solely by the provider

Locations for the Model and Runtime

Back-End intelligence uses cached (offline) results and send the results when needed to the user

- Good for finite number of predictions or low context changes; can invest time for difficult problems
- Cache results can be placed within services or deployed to the client
- Model updates might require invalidating all cached results

Hybrid approaches combine flavors of aforementioned approaches

- Quick response with client side intelligence while deep and accurate computation are triggered for a back-end intelligence
- Hard to orchestrate and maintain: What happens for inconsistent answer? How to synchronize results? What happens with result races?

Cloud Computing as Server-Centric Intelligence

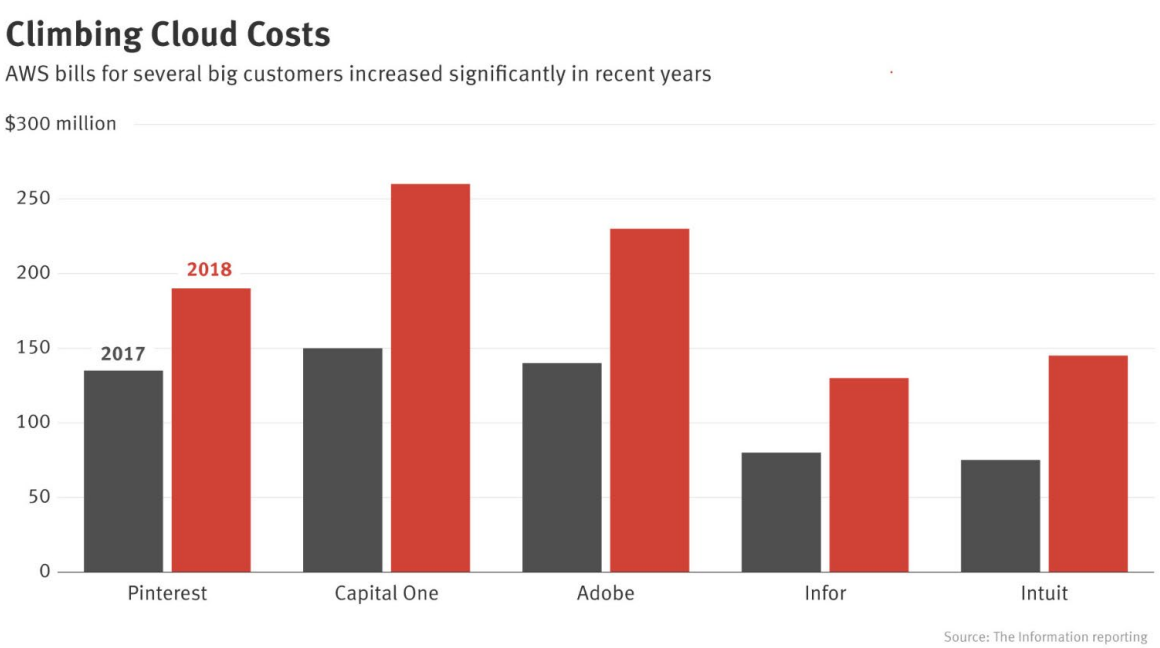
Idea: Do inference with powerful hardware via a Client-Server architecture.

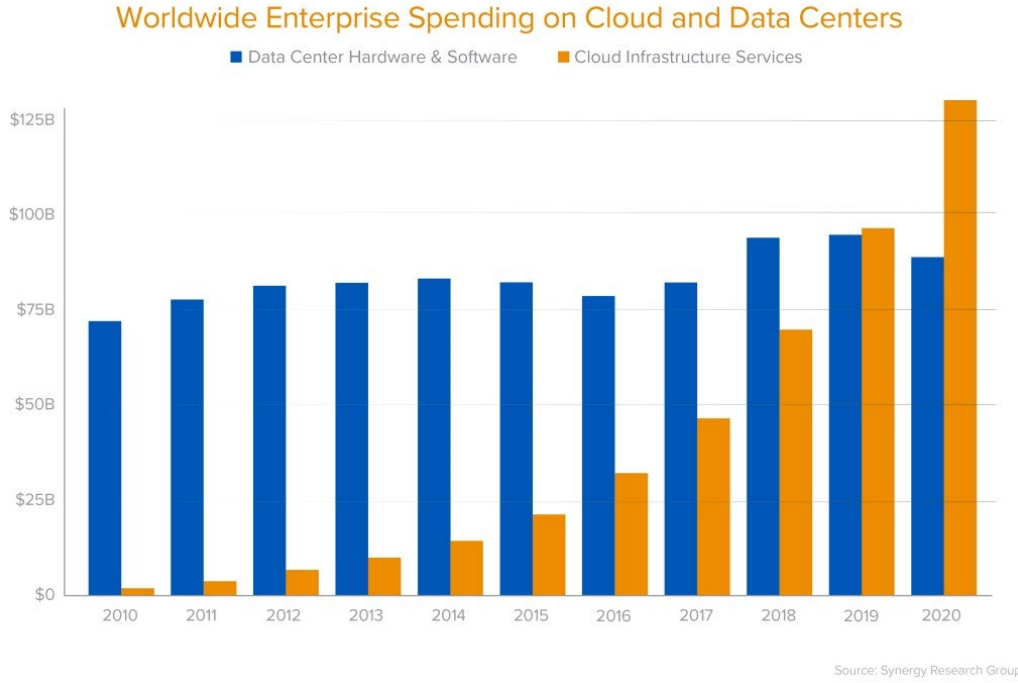
Examples: Speech processing (Alexa, Siri), face recognition, business predictions

Benefits:

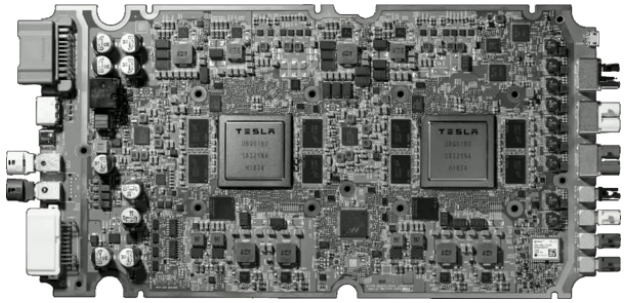
- Complex models: Inference on large models (GPT-3) only with powerful hardware
- Scalability: Easy to scale with more requests / customers to come
- Feedback: Easy to implement feedback loops, logging, and monitoring
- Productive: Production-ready cloud providers (good for startups)
- Reliable: Fall-back solutions, redundancy, and managed platforms
- Accessible: Reachable via API from the internet
- Security: Data is usually stored safely on a remote site with no physical access

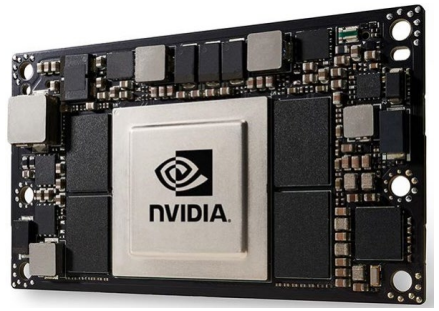
Always the Best Solution?





Edge Computing
as Client-Side Intelligence

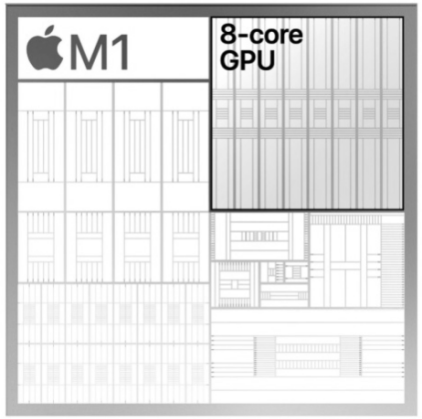




Idea: Do inference locally on the device that triggers the request

Examples: Local voice processing (wake word), local image improvement

Benefits:



- Privacy: No data needs to be sent to an external device / server
- Latency: No communication latency
- Independence: No requirement on internet connection
- Resilience: No dependence on external infrastructure
- Cost: No server infrastructure required (multiple cheap devices instead)

Question

Where would you put the intelligence?

Self-driving car?

Photo labeling?

Chat robot?

Smart light bulb?

Translator?

Face recognition?

Systematic Architectural Decisions

System landscape

Decide on …

Tools und runtime environment

Use all-in-one cloud providers or integrated (local) solutions

Intelligence location

Decide on …

Where the model should live (bringing runtime, context, and model together)

How to deliver predictions to the use case

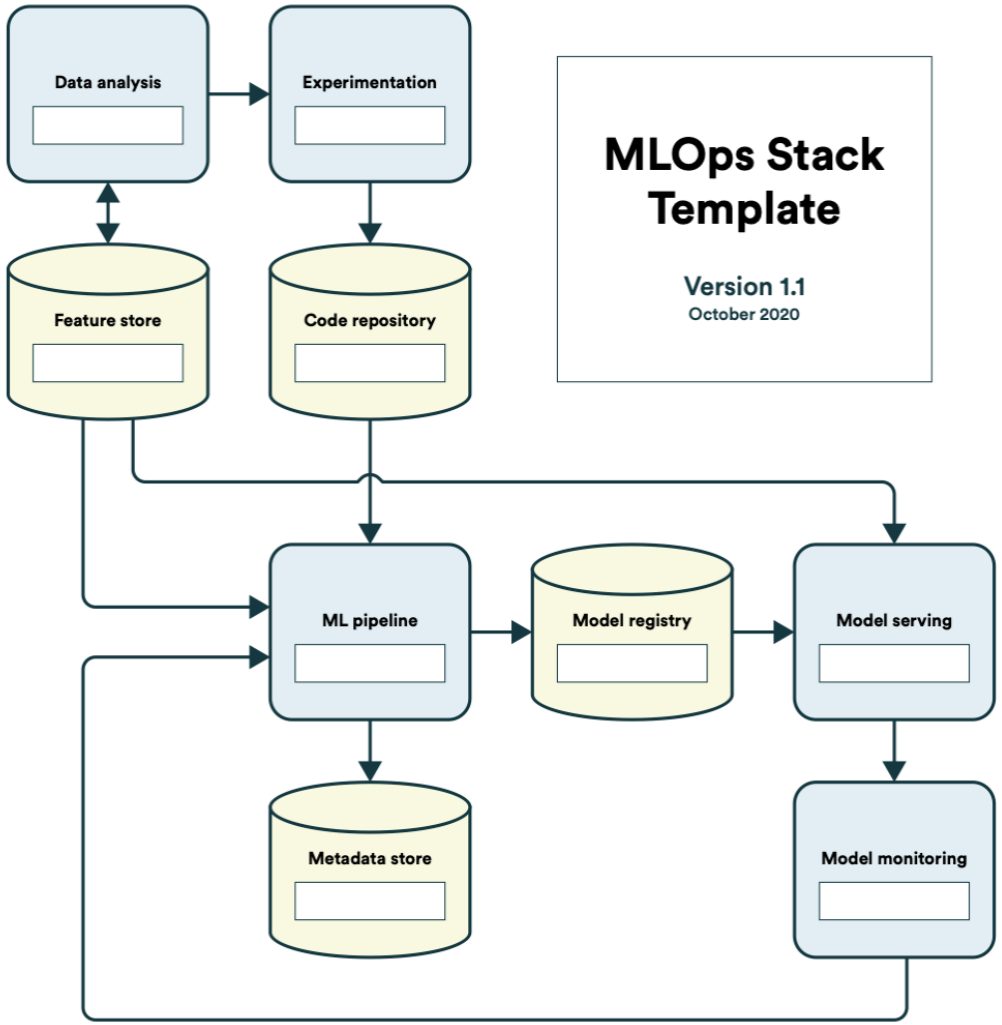
System architecture

Decide on …

System components and how they are connected

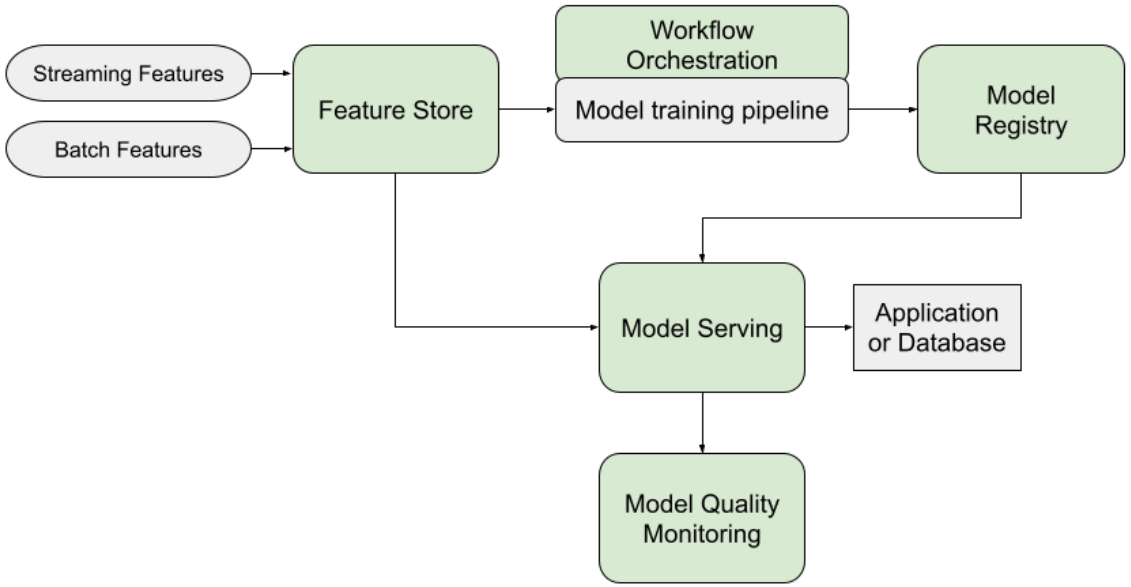
The degree of automation and scalability

ML Module Architecture (by ML-Ops.org)



MLOps Setup Components	Tools
Data Analysis	Python, Pandas
Source Control	Git
Test & Build Services	PyTest & Make
Deployment Services	Git, DVC
Model & Dataset Registry	DVC[aws s3]
Feature Store	Project code library
ML Metadata Store	DVC
ML Pipeline Orchestrator	DVC & Make

Architecture with Industry Realizations



	Model registry	Feature Store	Workflow Orchestration	Model serving	Model quality monitoring
Netflix	IH	Kind of*	Metaflow	IH	IH
Intuit	IH	IH	Argo Workflows	IH	IH
Intel	mlflow			Seldon Core	IH
Booking.com	IH	IH		IH	IH
Stitch Fix	IH	IH	IH	IH	
DoorDash	IH	IH		IH	IH
Uber	IH	IH	IH	IH	IH
Paypal	IH	IH	Airflow	IH	
Spotify	TFX ML Metadata	IH	Kubeflow Pipelines	IH	TF Model Analysis
Etsy		IH	Airflow	IH	IH
Pinterest	mlflow	IH	Airflow fork	IH	IH

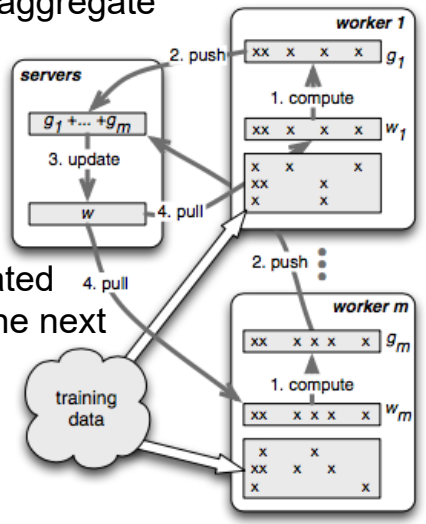
IH = In-house

Caveat: These are big companies and can effort (and require) in-house developments.

Parameter Server

ML models are basically a collection of weights (or parameters), which can go into the billions for modern models. Training needs to be distributed on several machines. Parameter servers enable distributed learning and inference.

Collect all weight updates and aggregate them



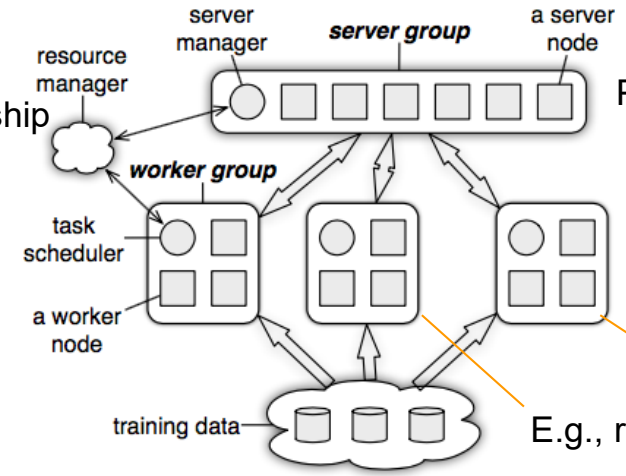
Get and update only the subset of features / weights that match the obtained training data

Pull the updated weights for the next training run

Distributed training

Provide parameters for multiple models; different synchronization schemes possible; fault tolerant

Liveness check, consistency, ownership assignment



Partition of feature space

One group per application

E.g., batch learning

E.g., real-time inference

Parameter server architecture

Flexible consistency modes

Graphics credit: Coinmonks @medium

Architecture Issues for Technical Debt

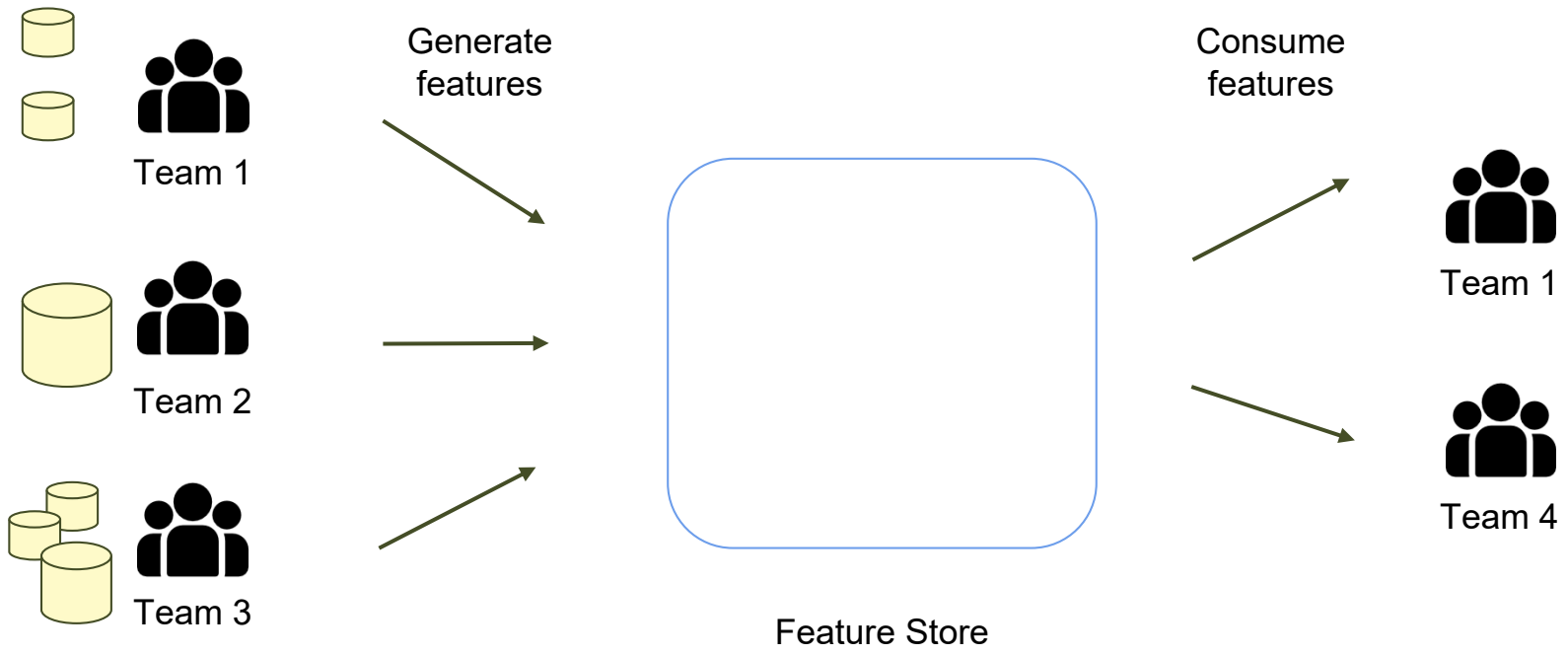
Issues reported by Google

- No principled way for accessing features during model serving
- Features cannot easily be reused among multiple MLOps pipelines
- Lack of collaboration, sharability, and reuse among ML projects
- Incosistency between features used in training and inference (serving time)
- Unclear how to know which features need to be recomputed when new data arrives (so entire pipelines needs to run for updating all features)

> *"Data is the hardest part of ML and the most important piece to get right. Modelers spend most of their time selecting and transforming features at training time and then building the pipelines to deliver those features to production models.* ***Broken data is the most common cause of problems in production ML systems***"
>
> Uber: https://eng.uber.com/scaling-michelangelo/

Feature Store

Idea: Store all ML features processed from raw data into a dedicated storage to be accessible from different teams at any time (extraction, training, inference, etc.)



(data engineers + scientists)

(ML scientists + SE)

Feature Store Benefits

Re-use and discover features across team and project boundaries

Access control and versioning support for features

Precomputation and automatic filling of features for entities of interest

Centralized place for data quality and integrity checks

Insurance of consistency of features in training and inference

Compute once, consume multiple times

Share expertise across teams

Easier to maintain quality of features

Easier to test

Better incorporation into MLOps pipeline

Better testability of ML models

Feature Store: Generate Entities

Product review = “....”

IDs

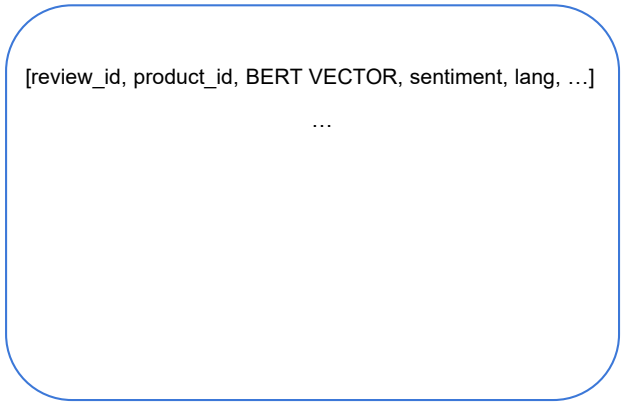
Feature vector

[review_id, product_id, BERT VECTOR, sentiment, lang, …]

Team 1 (e.g., NLP specialists)

Schema

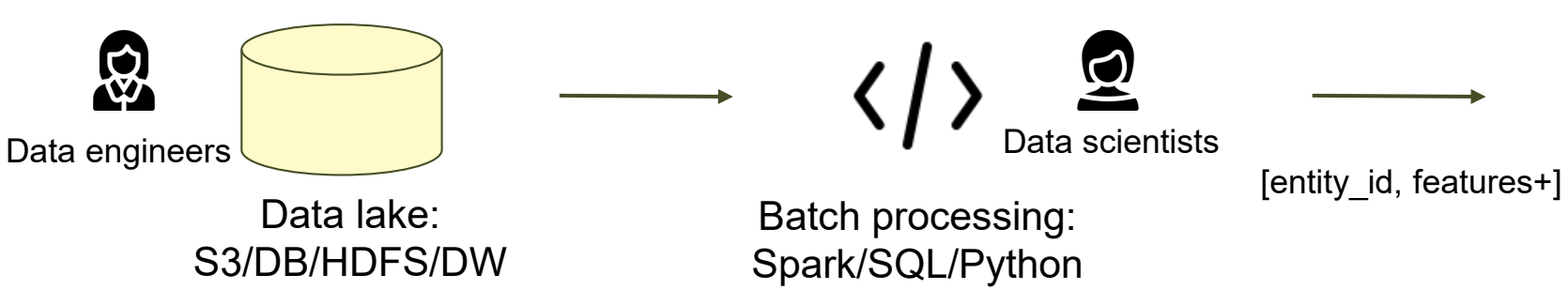
[entity_id, features+]



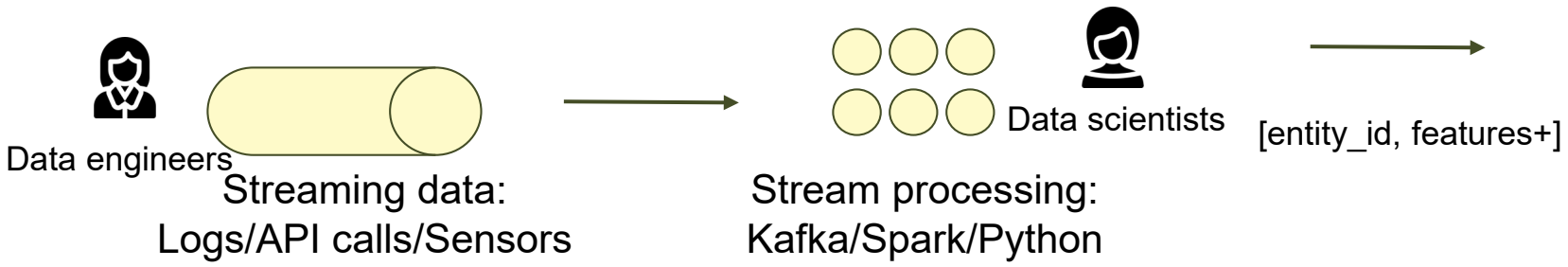
Feature Store

Feature Store: Ingesting Data

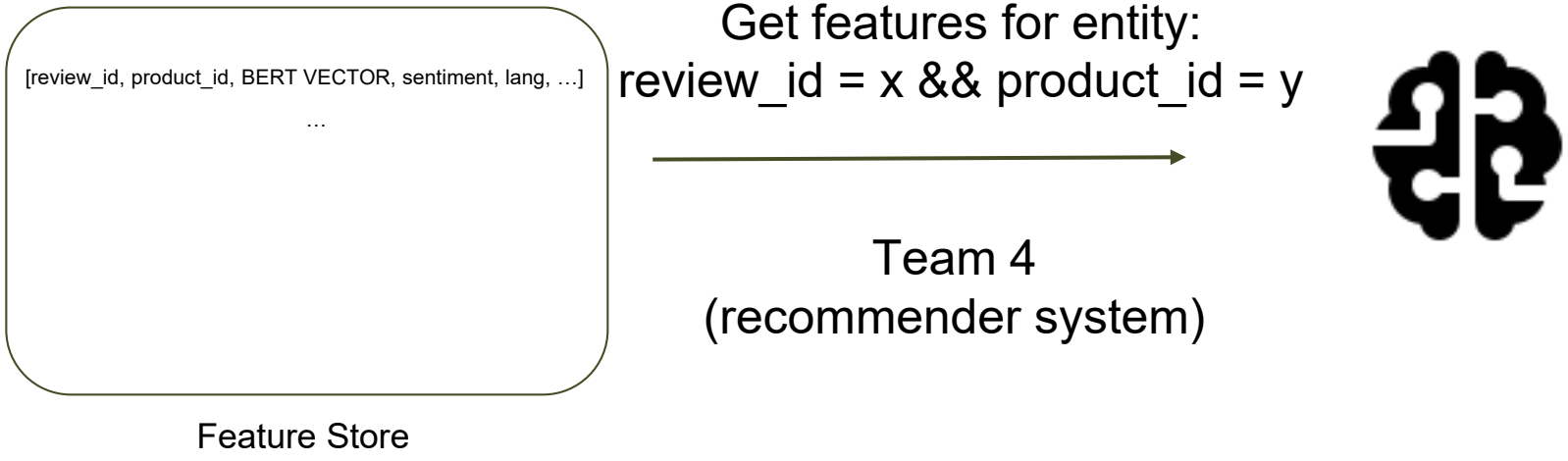
[1] Batch processing:



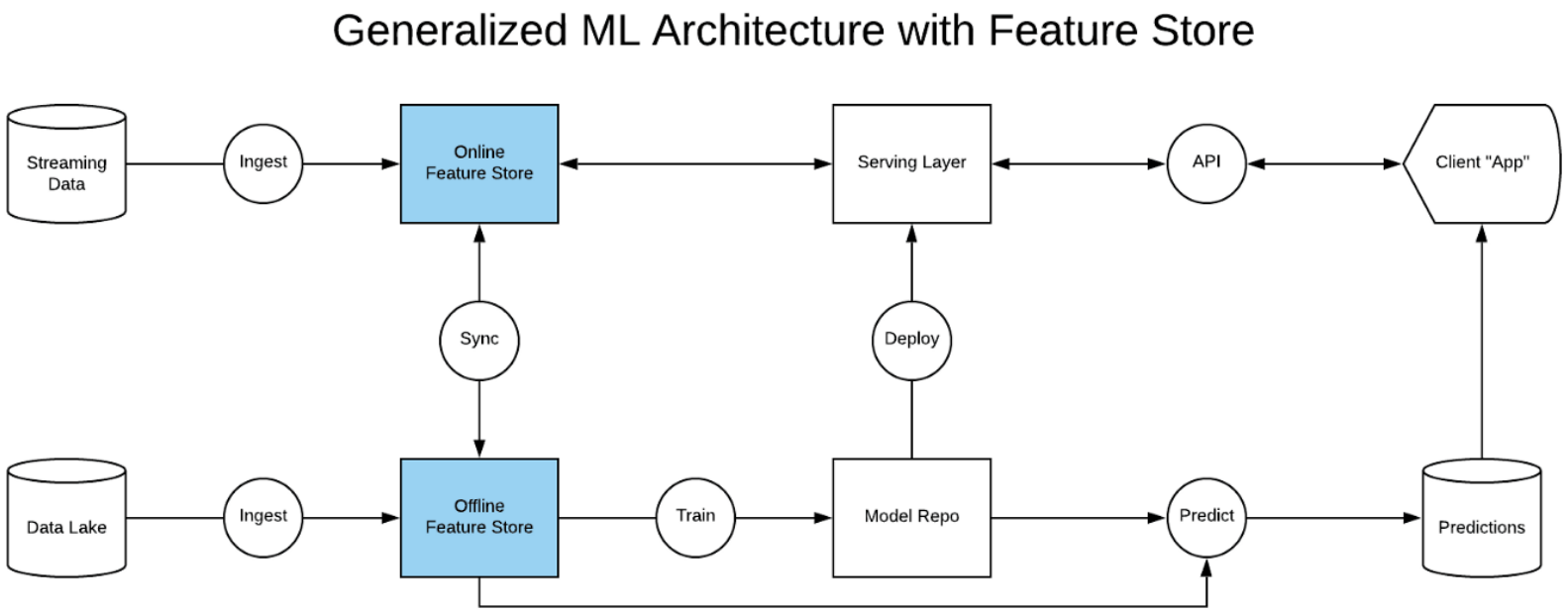
[2] Stream processing:



Feature Store: Consume Entities

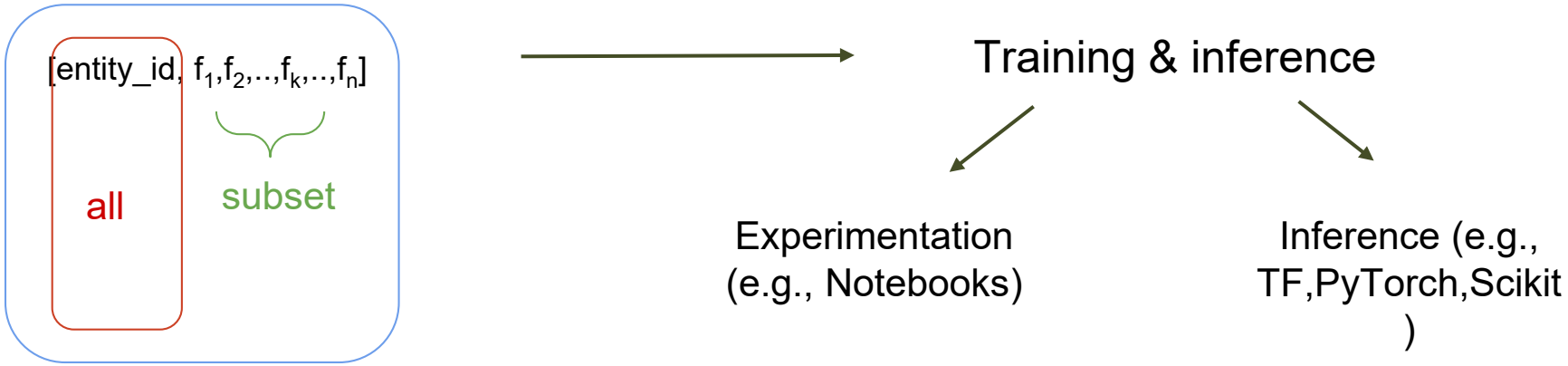


Feature Store: Consumer Variants

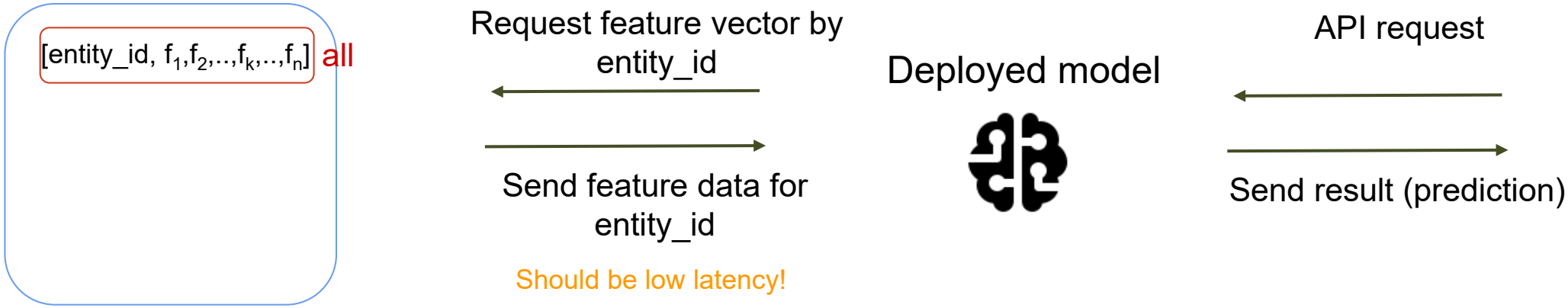


Feature Store: Consumption

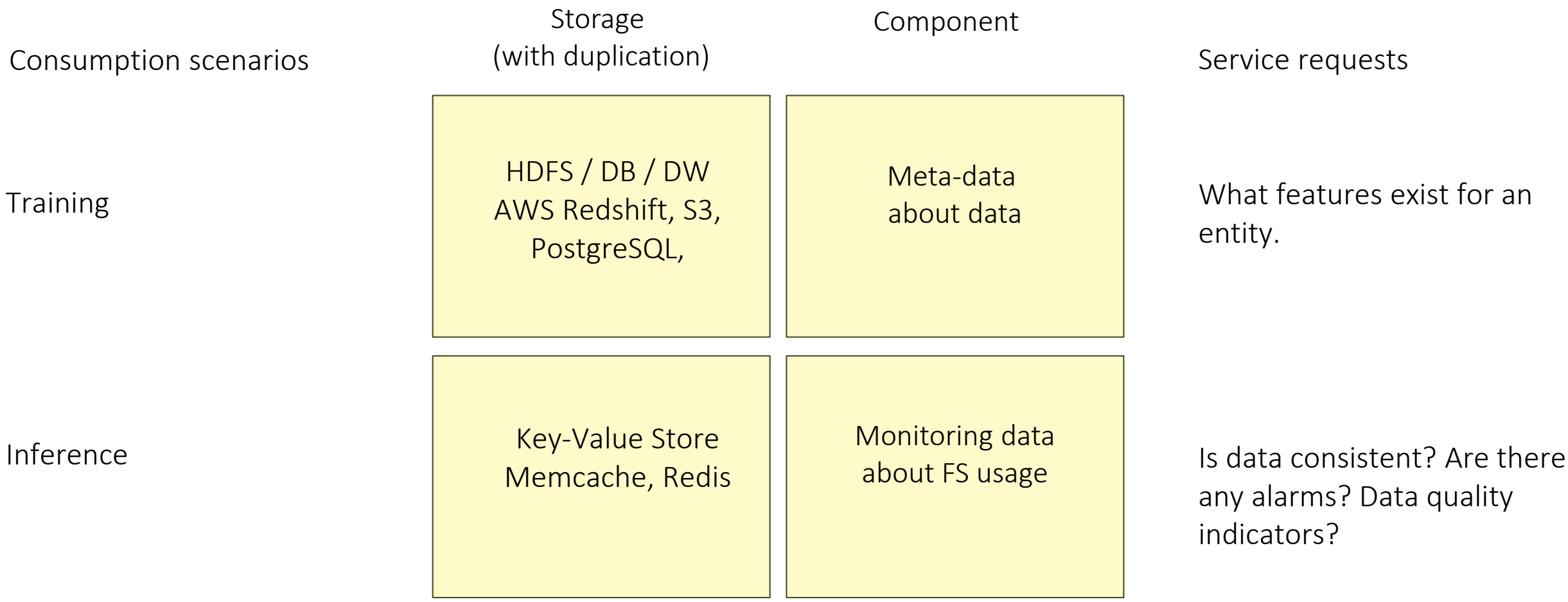
[1] Bulk load of feature subset:



[2] Request-based:



Feature Store: Internals

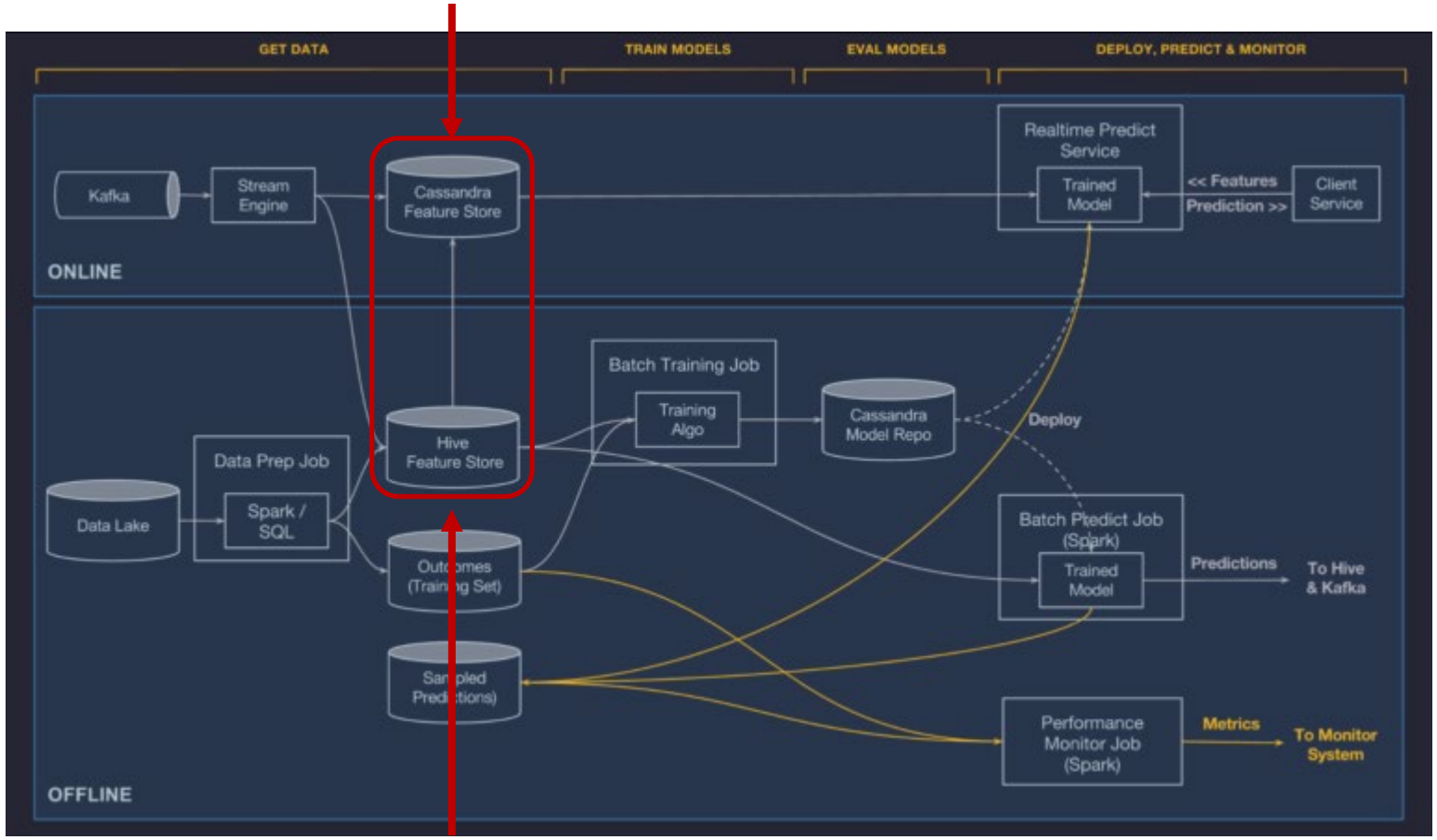


Feature Store at Uber (Michelangelo)

"Instead, we allow features needed for online models to be precomputed and stored in Cassandra where they can be read at low latency at prediction time."

Online pipeline

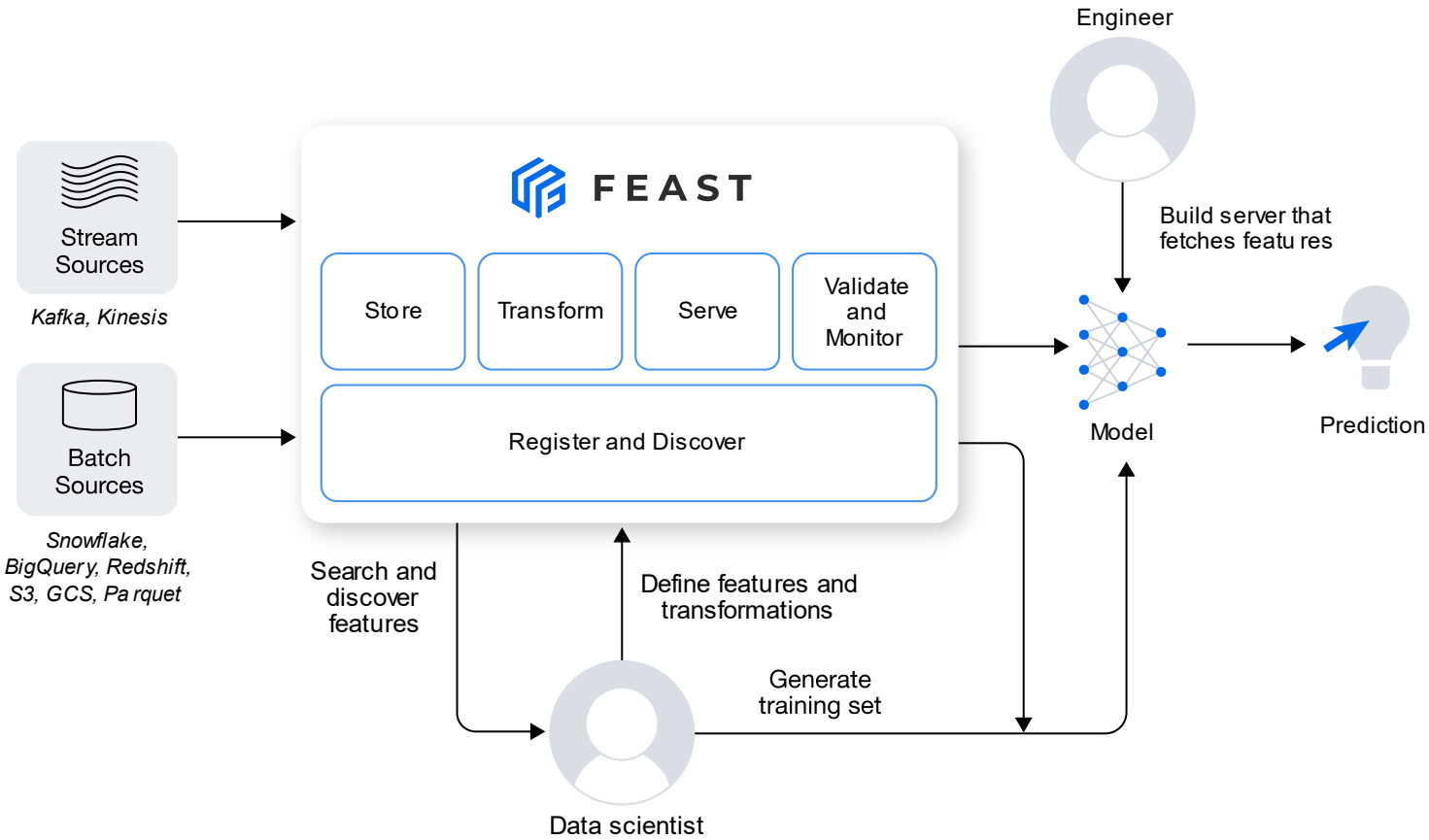
Offline pipeline



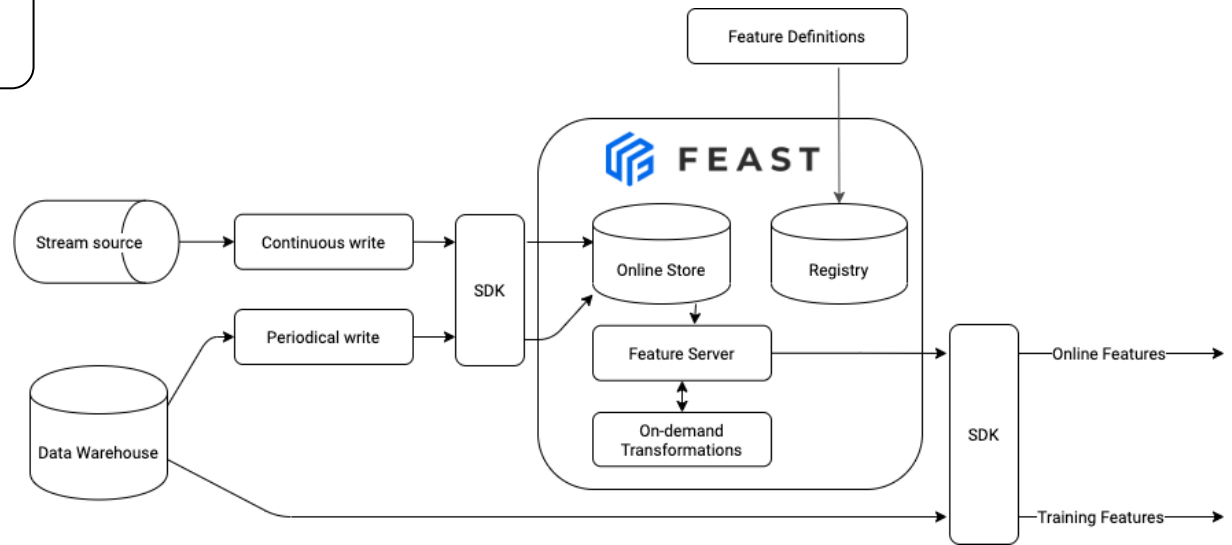
"We found great value in building a centralized Feature Store in which teams around Uber can create and manage canonical features to be used by their teams and shared with others."

At the moment, we have approximately 10,000 features in Feature Store…

Feature Store at Google/Open-Source: Feast

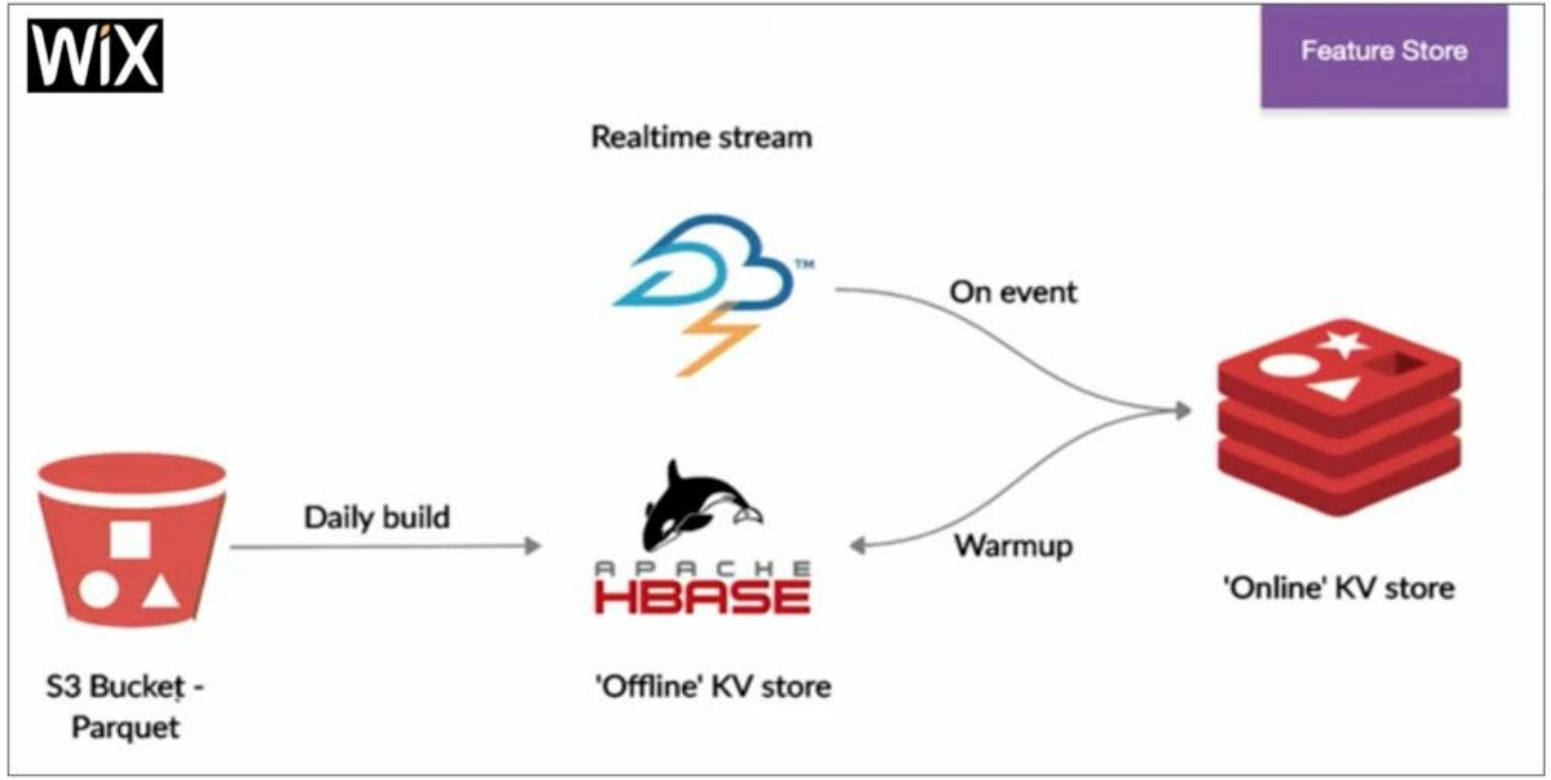


Open-source, can handle streaming data, very fast online processing, included in MS Azure Feature Store, feature vector can have defined life time, etc.



Feature Store at Wix

Over 90% of the data stored in the Wix feature store are clickstreams and the ML models are triggered per website or per user. Ran explains that for the real-time use cases latency is a big issue, and that for some of their production use cases they need to extract the feature vectors within milliseconds.



Once the system detects that a user is currently active, a 'Warmup' process is triggered and the features of that user are loaded into the online store (Redis) which is much smaller than the offline store since it holds only the user history of the active users. This 'Warmup' process can take several seconds. And finally, features in the online feature store are continuously updated using fresh live 'real-time" data directly from the streaming sources per each event coming from the user (using Apache Storm).

Commercial Feature Store: Tecton

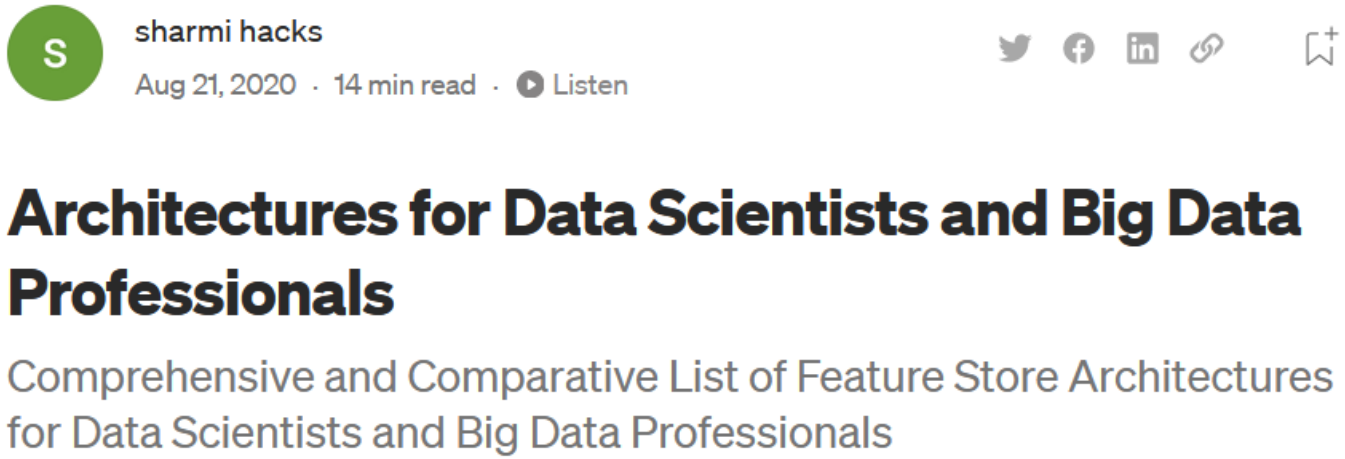
Like with Feast and the Wix Feature Stores, Tecton also defines the features in the registry so that the logical definition is defined once for both offline and online stores, significantly reducing training-serving skew to ensure high accuracy of the ML model also in production.

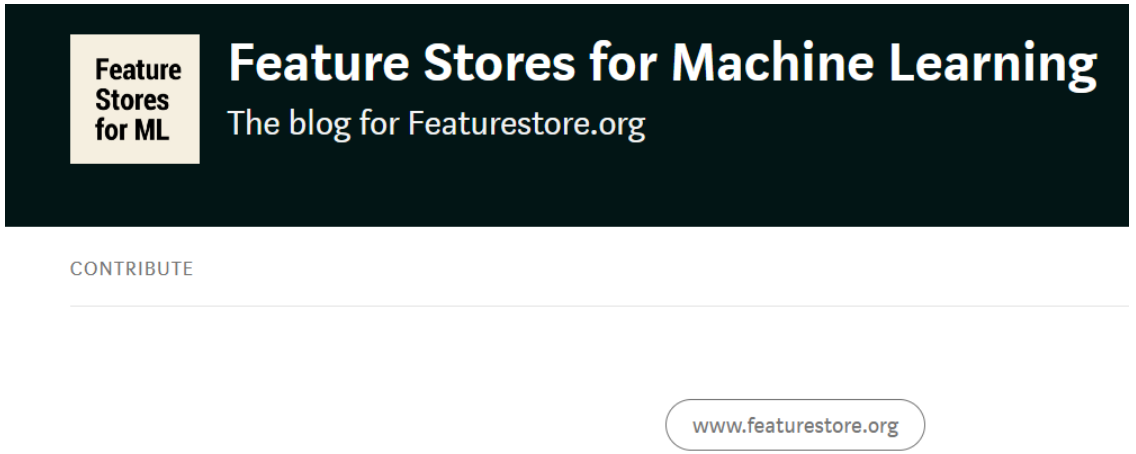
Commercial Feature Store: Hopsworks


```python
from hops import featurestore
raw_data = spark.read.parquet(filename)

polynomial_features = raw_data.map(lambda x: x^2)

featurestore.insert_into_featuregroup(polynomial_features, "polynomial_featuregroup")
```

```python
from hops import featurestore

features_df =
featurestore.get_features(["average_attendance",
"average_player_age"])
```

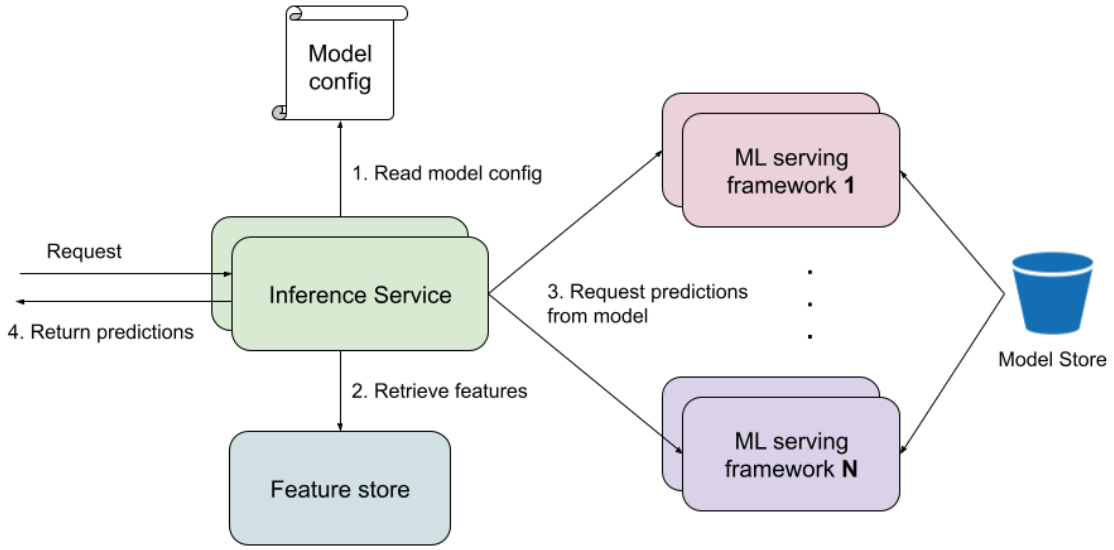
More Sources:





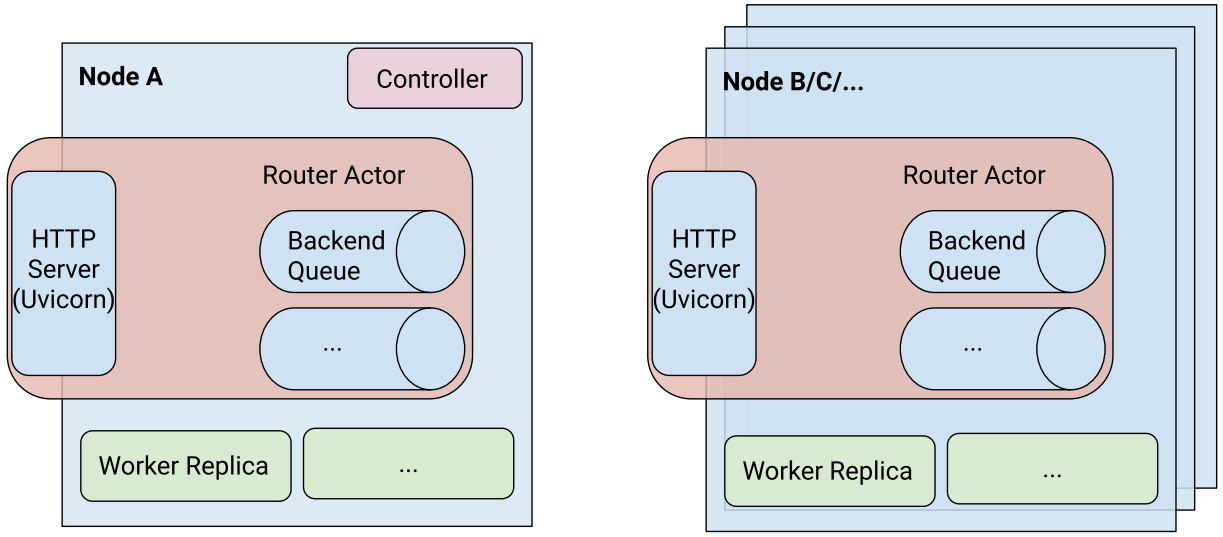
Model Server / Model Serving

Model servers or serving platforms enable to store, version, and serve up to hundreds of models simultaneously. This way also complicated ML pipelines, such as stacking and ensemble learning can be realized. Enables other deployment scenarios, such as shadow modes, A/B testing, etc.



Model Server: Ray Serve

Ray Serve enables composing multiple ML models into a deployment graph. This allows you to write a complex inference service consisting of multiple ML models and business logic all in Python code.




```python
from fastapi import FastAPI

app = FastAPI()

@serve.deployment
@serve.ingress(app)
class Counter:
  def __init__(self):
      self.count = 0

  @app.get("/")
  def get(self):
      return {"count": self.count}

  @app.get("/incr")
  def incr(self):
      self.count += 1
      return {"count": self.count}

  @app.get("/decr")
  def decr(self):
      self.count -= 1
      return {"count": self.count}
```

Language Chains

Systematic Architectural Decisions

System landscape

Decide on …

Tools und runtime environment

Use all-in-one cloud providers or integrated (local) solutions

Intelligence location

Decide on …

Where the model should live (bringing runtime, context, and model together)

How to deliver predictions to the use case

System architecture

Decide on …

System components and how they are connected

The degree of automation and scalability

Topic III:
Design Patterns and ML-Specific Components

Best Practices for Uncertain Decisions

Do nothing!

Postpone decisions or developments (e.g., whether to support batch processing or automated learning) until the requirement becomes clear

Best Practices for Uncertain Decisions

Implement the simplest solution!

You proceed with the project and can replace the functionality later. For instance, implement HTTP Basic Auth quickly and replace it later, when it is clear what is actually needed.

Best Practices for Uncertain Decisions

Design for multiple options!

Create abstractions and modules (via interfaces) to be able to react to changes. For instance, implementing a model in a container will make it independent of the cloud provider.

Use standards / de facto standards

Standards increase compatibility and interoperability with other tools and frameworks. For instance, use standard file formats, such as JSON for data transfer or ONNX format for the model.

Best Practices for Uncertain Decisions

Make documented assumptions

Decision record template by Michael Nygard

This is the template in Documenting architecture decisions - Michael Nygard. You can use adr-tools for managing the ADR files.

In each ADR file, write these sections:

Title

Status

What is the status, such as proposed, accepted, rejected, deprecated, superseded, etc.?

Context

What is the issue that we're seeing that is motivating this decision or change?

Decision

What is the change that we're proposing and/or doing?

Consequences

What becomes easier or more difficult to do because of this change?

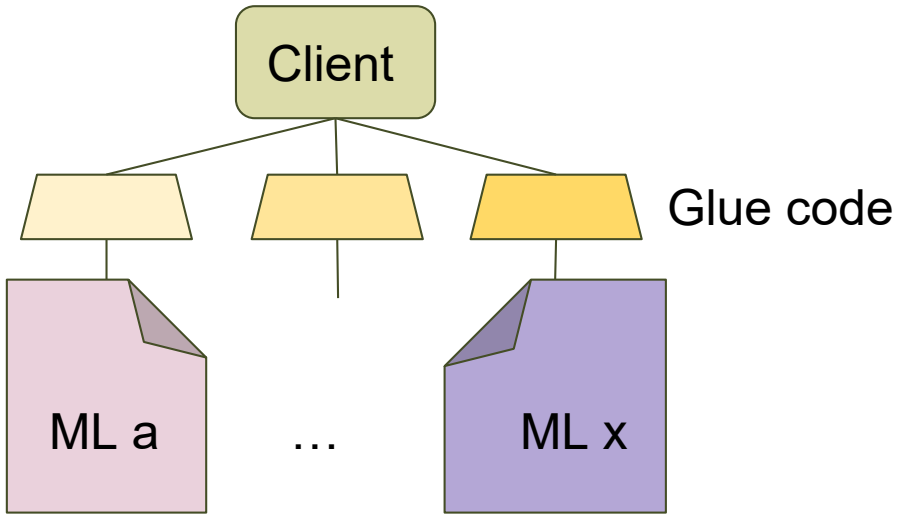
Write down your assumptions on which basis you make a decision. Review and revise this assumption and decision later. Use, for instance, the artifact decision record (ADR):

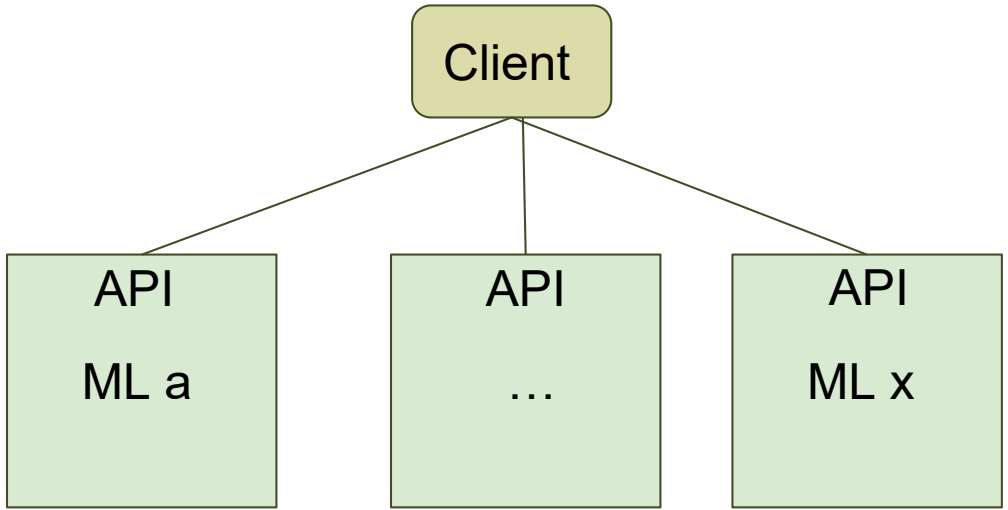
https://github.com/joelparkerhenderson/architecture-decision-record

Wrap Black-Box Packages into Common APIs

Problem: ML frameworks and libraries are developed as stand-alone entities. Integrating them into a single system needs glue code for connecting them and providing them with the necessary data. With increasing tools and frameworks, the glue code increases and becomes hard to maintain, to test, and to extend.

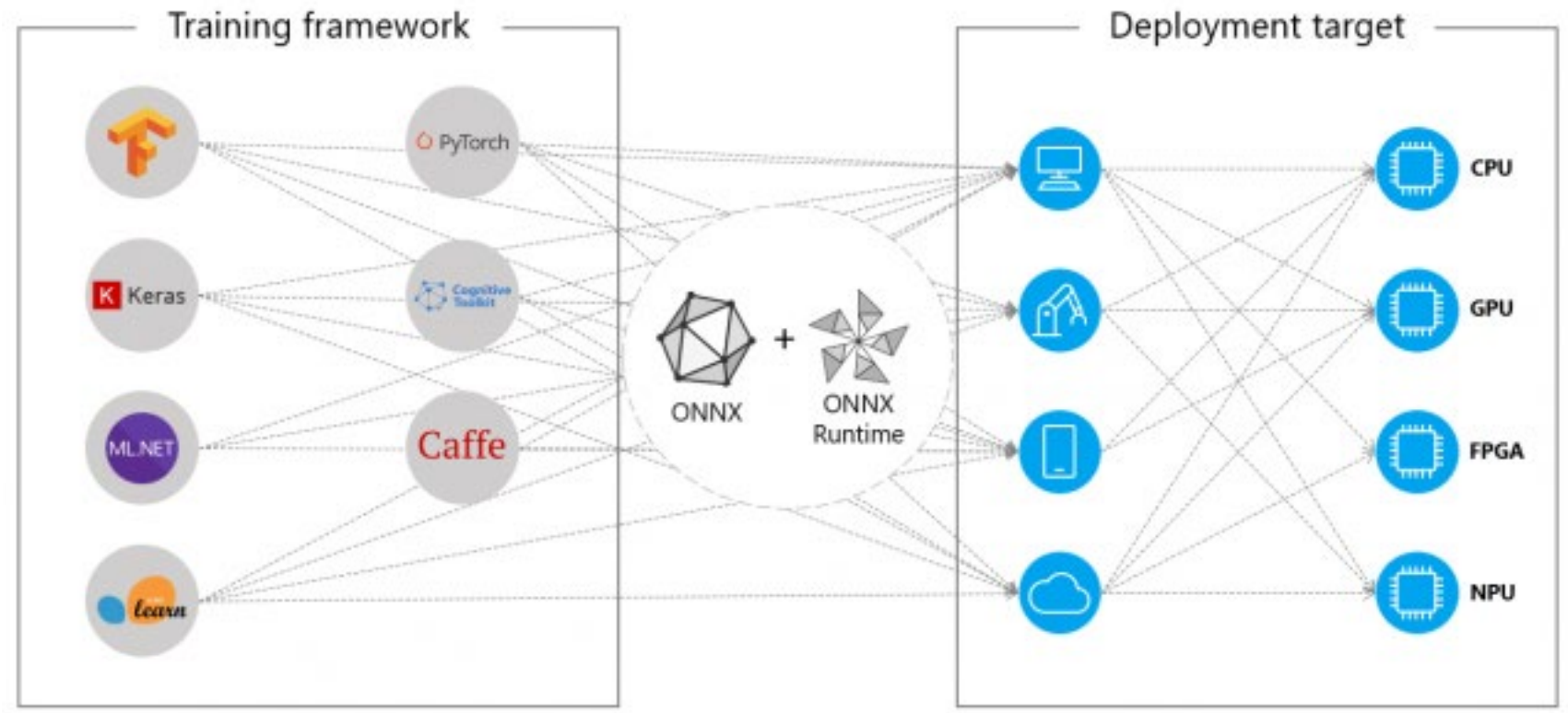
Solution: Wrap domain-specific packages into common APIs, such that infrastructure code can be reused for different projects no matter what is “behind” the API.





Keep Flexibility via Standards

To maintain flexibility for deployment targets and architectural freedom, use representation formats that are interchangeable, such as ONNX for models



Limit the Number of Languages

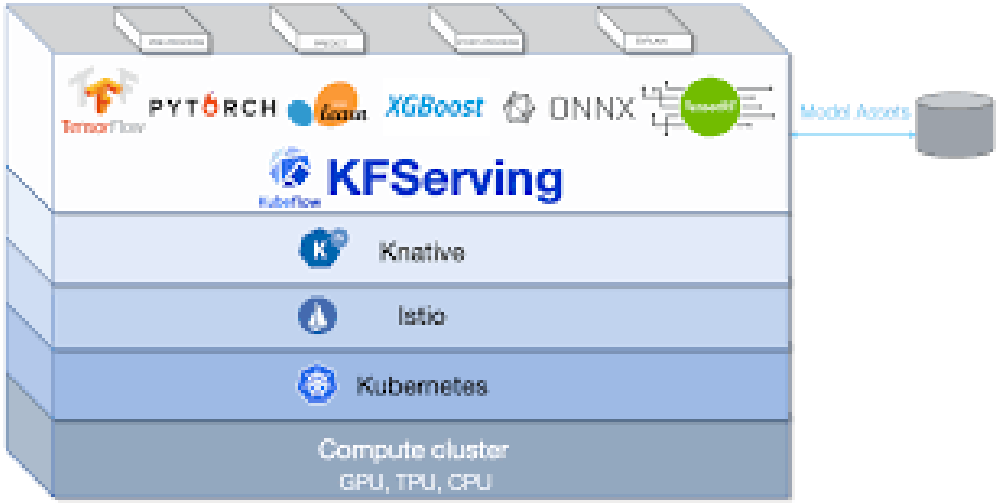
Problem: With different languages offering better support for specific tasks, it is tempting to build an architecture out of a diverse set of languages. However, the effort for testing such a heterogeneous landscape is high and transfer of ownership becomes problematic.

Solution: Limit the use of languages to improve interoperability and knowledge sharing (e.g., keep Python for ML tasks and Java for SW / serving tasks instead of Python + Julia + Java + Goo).

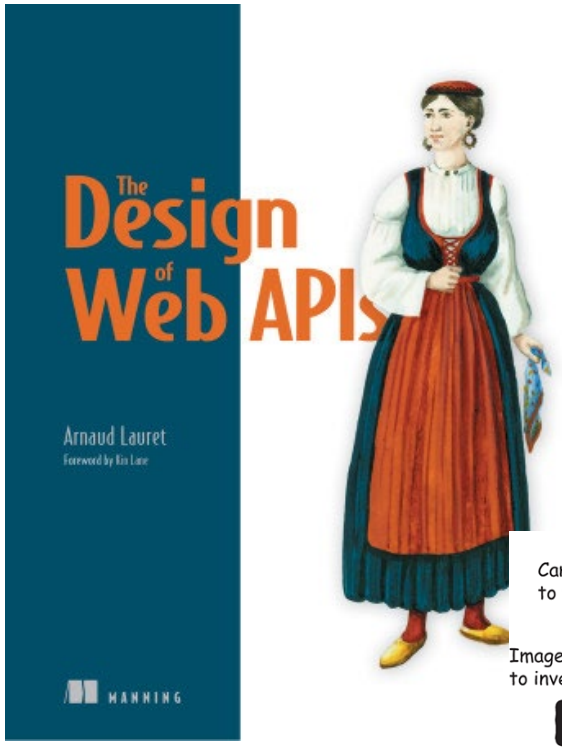
Containerize your Pipeline

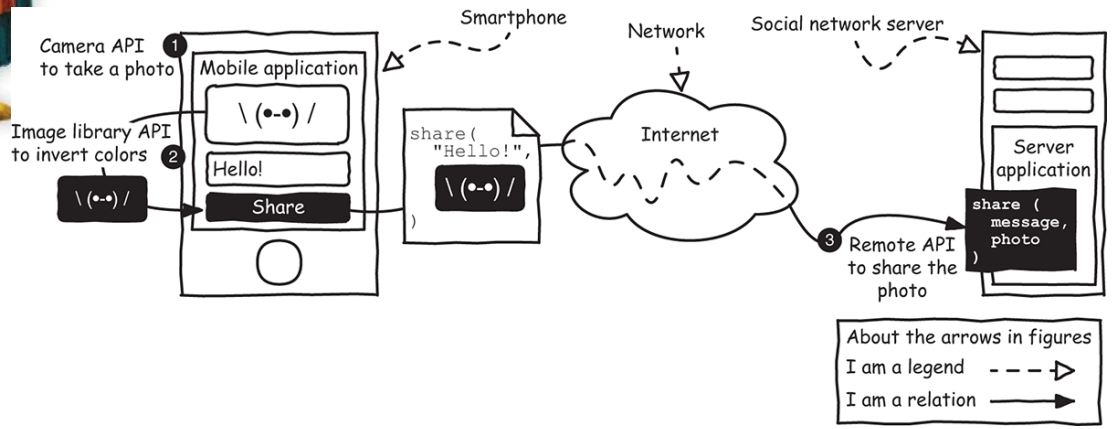
Problem: From data source till deployment, the process runs through different steps (collection, cleaning, feature extraction, model definition, training, deployment). We may want to execute in certain situations only parts of the pipeline or exchange them. Hard coding the interfaces for these steps makes this problematic.

Solution: Encapsulate every major step into an own container and connect the containers through a pipeline (e.g., using KubeFlow). Enables separate logs and error analysis, replacement of containers, distinct resource allocation, etc. Produced artifacts stored in extra location to be analyzable later.



API Design





http://apistylebook.com/design/guidelines/

How to start? Focus on interfaces

Level 2 and 3 of C4 model: view the ML component as black box and define

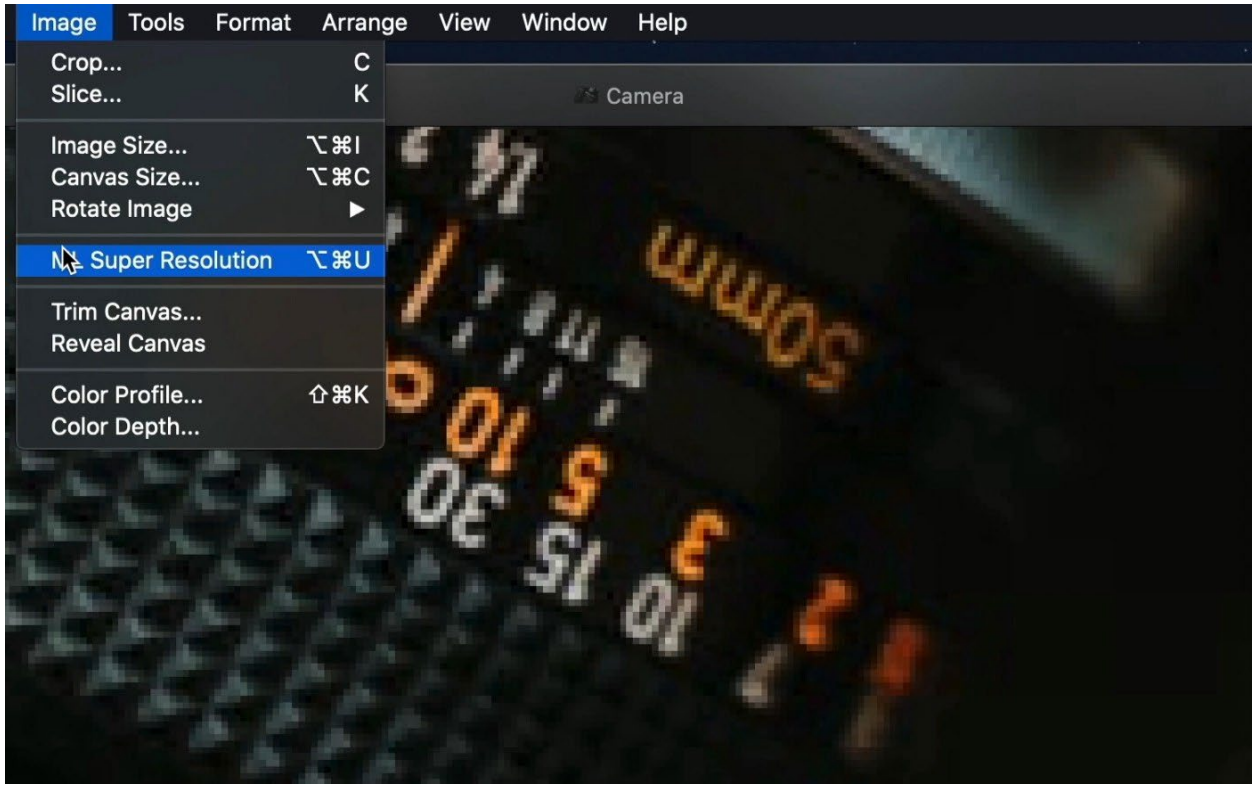
- How data is fed into the component (protocol, format, data description)
- How clients connect to the component (authentification, API)
- How the component interacts with external services (API, protocol, messaging)
- How an optional internal lookup service discovers the component (load balancer, infrastructure)
- Service Level Agreements (SLAs) to document non-functional requirements (availability, throughput, performance)

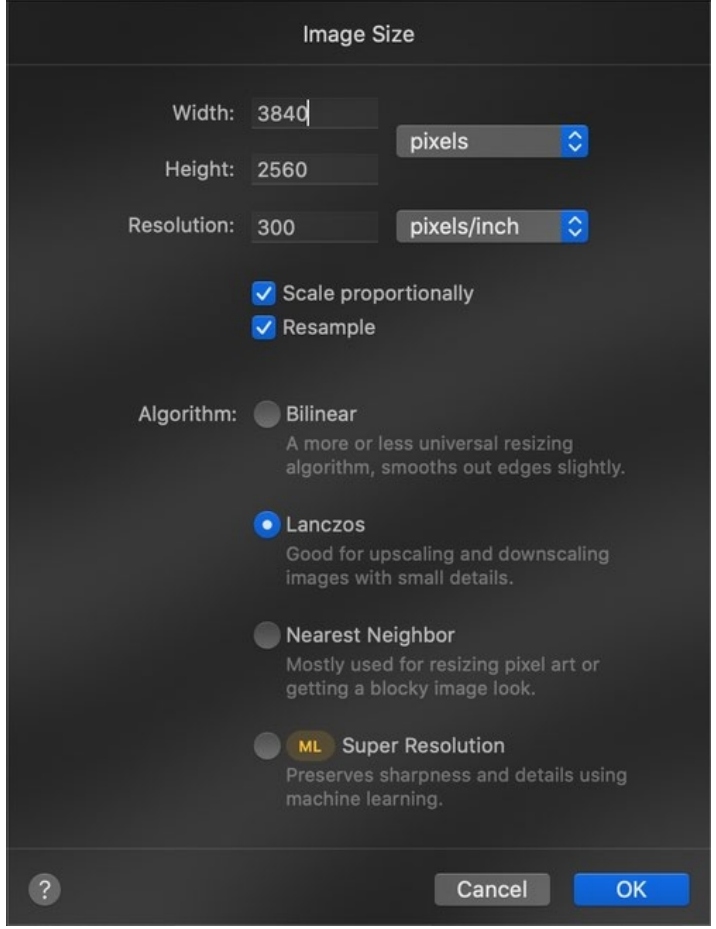
Build component with the specification and mockups to begin testing the interface

Topic IV:
UX-ML Design Principles

Do we need different user interfaces for interacting with ML systems?

Example: Super Resolution via ML

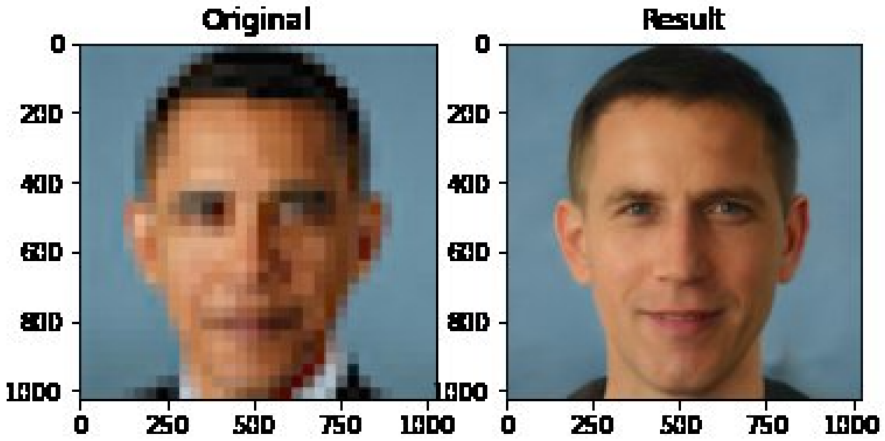




UI suggests that the ML algorithm works the same way as the other algorithms

Scaling algorithms interpolate among the existing pixels in a deterministic, content-agnostic way.
But, what about ML algorithms?

Example: Super Resolution via ML



Are ML algorithms deterministic in its output?

Are they content-agnostic?

Are they free of bias?

A good UI would make the user aware of this non-deterministic, possibly biased process.
A good UI would possibly provide multiple outputs and let the user select the desired one.
A good UI would clearly separate this kind of resolution improvement from the other algorithms.
A good UI would … [name it]

How to represent intelligence?

Internally, we do something like that: `prediction = model.getPrediction(context)`

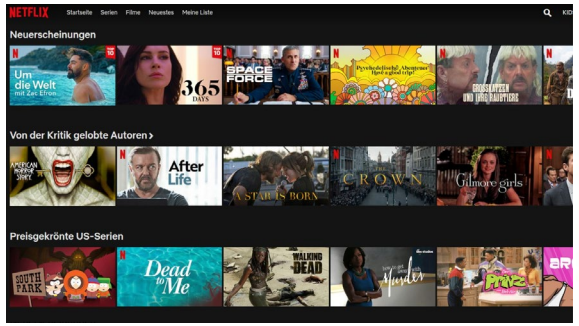
How to present this result to the user, software component, or service in need?

Factors when presented to machines and software components:

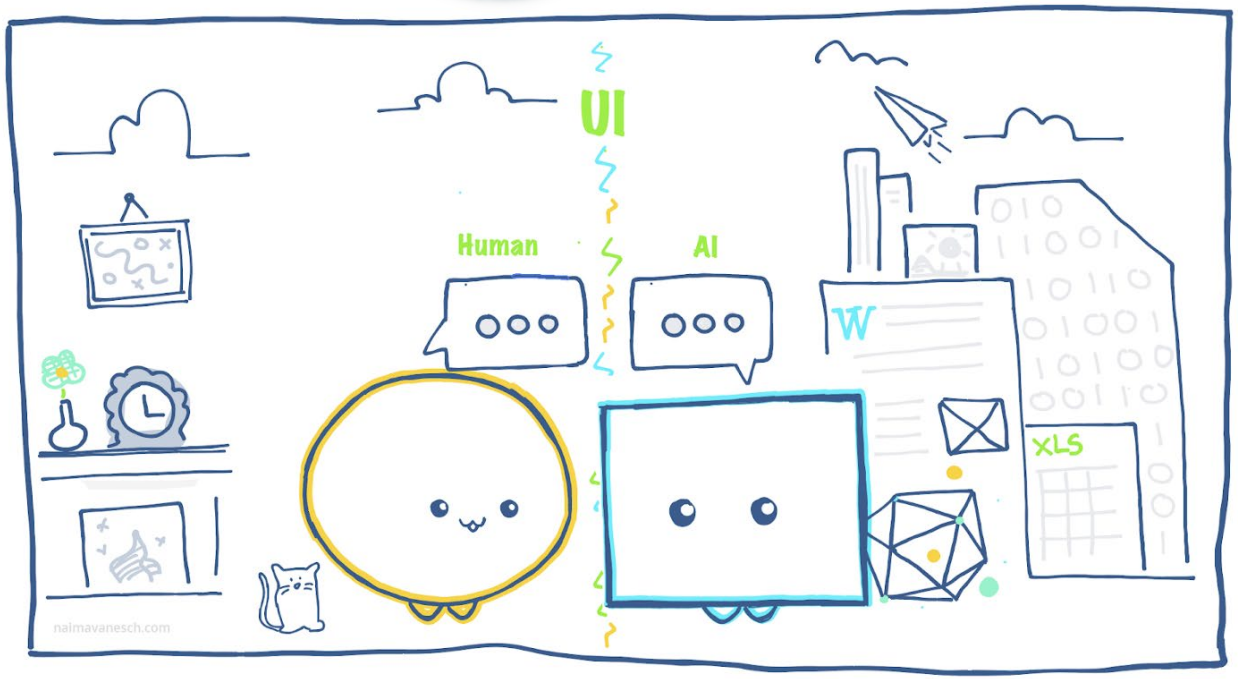
- Easy to process and manipulate
- Match the target API / language

Question: Can you name some AI-Enabled Interfaces?

Netflix



iRobot Roomba



Naïma van Esch

Amazon Alexa

Nest Thermostat

AI-Enabled UI

UI depends on the AI and the channel of communication

Examples:

- Voice interaction
- Embedded in a vacuum cleaner
- Search engine
- Playlist browsing

Factors when presented to humans:

- Minimize chance of mistakes and misunderstandings
- Make the intelligence easy to manage and interpret
- Target the correct audience (experts vs. novices vs. non-technicals)
- Easy to use
- Simple and efficient

Design Principles for AI-Enabled UI

1. Discovery and expectation management,
2. Design for forgiveness,
3. Data transparency and tailoring, and
4. Privacy, security, and control.

Disclaimer: There are multiple principles. We review some of them as they overlap and we strive for a general overview instead of a comprehensive review of the field.

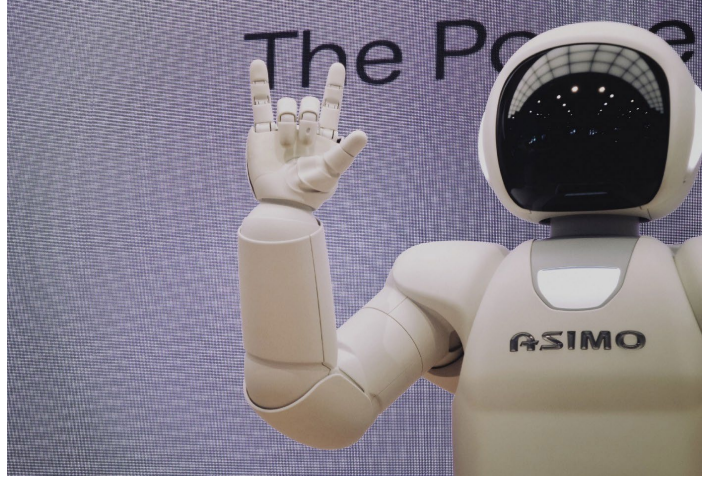
Discovery and Expectation Management

Idea: Set user expectations to avoid false expectations

Realization:

- Make users aware about the capabilities of the tool (do not oversell AI capabilities, be precise in what it can do)
- Gain most out of AI with little input as possible (require only „natural input“ --- feels natural to the user; should accomplish the user‘s goal is an easy and efficient way)
- Prepare for unexpected tool usage (users will use the AI in novel ways)
- Prepare users for AI mistakes (be honest and inform users about possible flaws)

Design for Forgiveness



Idea: Design UI such that users are inclined to forgive AI failures

Realization:

- Imitate others objects / creatures where humans are inclined to forgive and so the AI (compare that with Siri or Alexa.. Are you so forgiving?)
- Delightful features increase likelihood for forgiveness (e.g., incorporate humor, greeting vacuum cleaner when coming home)
- Do not require internet connectivity (can you maintain a basic functionality?)

Data Transparency and Tailoring

Idea: Be transparent about data collection and make it customizable

Realization:

- Report honestly about the data the AI collects from the user
- Design UI to enable user inputs to the AI (goal of relearning via feedback and achieve the flywheel)
- Give users the ability to adjust what the AI has learned (remove data sets / history information from the learning process)

Privacy, Security, and Control

Idea: Gain trust via privacy, security, and direct control over AI features

Realization:

- Design security measures, especially for AI using personal data (two-factor authorization, biometric login, encryption of application data)
- Prove promised functionality (e.g., via test runs for scheduled activities)
- Make AI actions interruptible, so user can take over (stop buttons for auto mowers, take over controls for autonomous driving, muting microphone input for alexa, etc.)
- AI should have a permission system and act on it (asking for permission rights for consequential actions)
- AI must notify users in case of errors (be clear when inputs are missing or needed, and suggest a possible fix

UX Design Decisions for Apps

Proactive vs. Reactive

Visible vs. Invisible

Dynamic vs. Static

Explicit vs. Implicit feedback

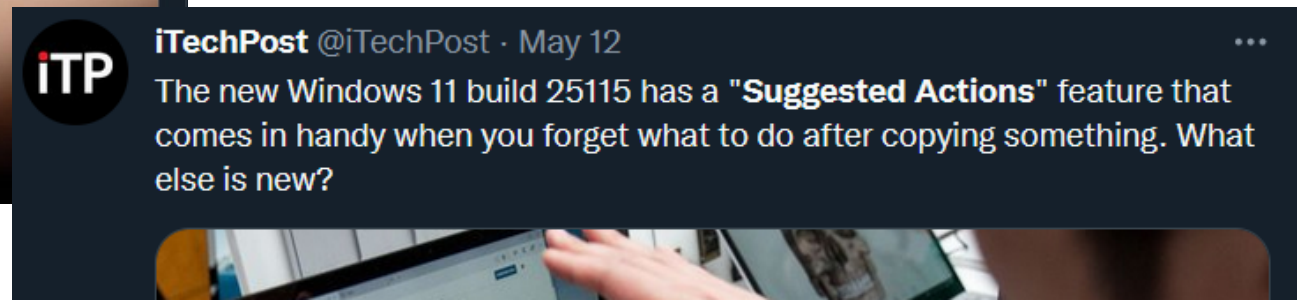
Apple UI/UX app guidelines

AI Usage Scenario: Proactive

Proactive

- Provide output of model to the user without request
- Prompt new tasks, additional information, or engagement
- Examples: Shortcut suggestions via driving
- Negative example:

- On copying a date and/or time, Windows will pop up an inline light dismissible UI that suggests creating the event using supported calendar apps. Upon user selection of preference, the app is launched with its corresponding calendar event creation page with auto filled date and/or time.



AI Usage Scenario: Reactive

Reactive:

- Trigger via an action or query
- Example: Suggestions via Typing

Since proactive AI actions have not been asked for, there is the danger of being annoyed and people getting easily frustrated by false results/suggestions or low information.

Circumvent by user tests (what is the right frequency of proactive behavior and what information is deemed useful) and enrich with further features and data.

UI Style: Visible vs. Invisible AI

Visible AI shows (processed) model outputs to the user, such as recommendation list, suggestions, numbers, etc.
It allows the user to interact with the AI features (e.g., offering feedback).

Enables transparency of the AI results.

AI actions not obvious to the user, for example, by adjusting the tap area of keys over time.
Hard to provide feedback and hard to value the AI actions.
Dangers of an intransparent decision process.

Dynamic vs. Static AI

Can the AI improve dynamically (e.g., FACE-ID, tap area) or statically through updates?
This has consequences on the required updated frequency, training capabilities, and data gathering functionality of the app.

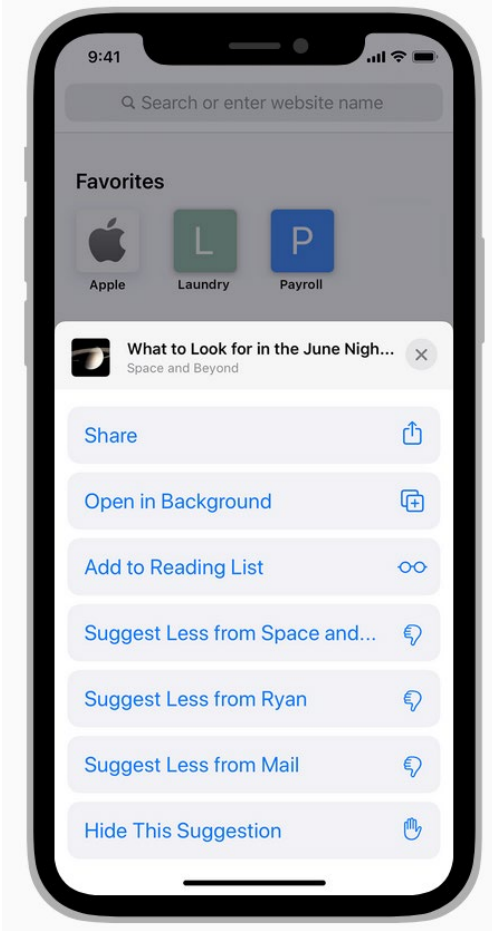
Dynamic AIs may require other software features, such as calibration and feedback.

Explicit Feedback

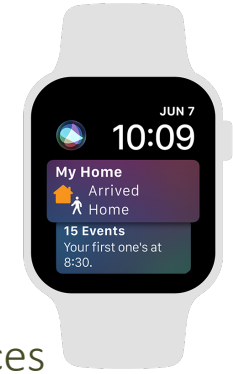
Supervised ML models need continuously labeled data, so why not getting it directly from the users in the wild?

Realization:

- Feedback directly provided by the users
- Requires UI controls tailored to the model output and functionality to collect and send the feedback to the server
- Favoriting and emotions are typical explicit feedbacks, but for social platforms. Apps can monitor them to obtain implicit feedback
- Explicit feedback should only be requested when necessary
- Make explicit feedback always optional
- Don‘t provide simultaneously options for both positive and negative feedback (good suggestions should be standard, so ask only for mistakes)
- Use simple and direct language to explain explicit feedback and consequences (e.g., suggest less pop music); Icons may help, but alone are not always clear
- If applicable, provide multiple, detailed options for explicit feedback
- Act on explicit feedback and store it (e.g., deleting certain photos immediately)
- Consider explicit feedback also for when and how to show results, not only what



Implicit Feedback



Feedback that the software can log from a user's behavior with it. Wide range of information from user preferences over activities. Enables to improve experience without any additional work by the user.

Realization:

- Secure user information as implicit feedback often contains sensitive information (privacy issue!)
- Control the degree of information (explain users how the app obtains and shares information and restrict it)
- Do not constrain exploration (implicit behavior may reinforce user behavior, but also constrain app usage to a limited subset, e.g., suggest always the same favorite actions)
- Use multiple feedback signals to improve model output and mitigate failures (avoid misinterpretation of user activities by combining multiple feedback signals)
- Withhold private or sensitive information (implicit feedback with privacy concerns should not be incorporated in suggestions, e.g., a shared account could exposure private information)
- Use feedback to place AI actions at expected time intervals (word suggestions during typing vs. Song recommendation via searching)
- UI changes in the app may can changes in implicit feedback (if interpreted automatically, this can cause to erros)
- Confirmation bias is a threat (you only get feedback about what you provide and not about what else is there)

UI/UX Design Principles for Model Output: Mistakes

AI models will always make mistakes, so anticipate and prepare accordingly.
Here are some principles:

Understand significance of a mistake‘s consequences: Show corrective actions and provide help for serious consequences

Make it easy to correct mistakes: Allow users to correct errors

Continuously update ML features to reflect changed user behavior: Use implicit feedback to discover behavior changes; use domain-specific general features that are timely

UI/UX Design Principles for Model Output: Multiple Options

Presenting multiple results can create a sense of control for the user; manages also expectations. Applicable for proactive features, requested options for a queried result, possible corrections to fix AI mistakes

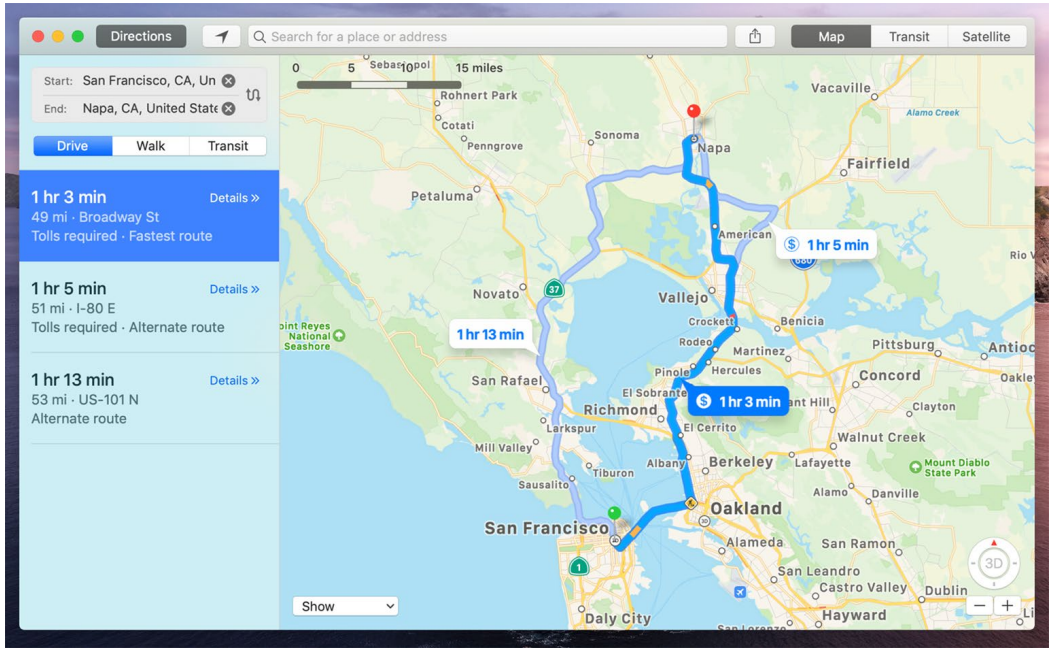
When providing multiple options, design for:

Diverse options (balance accuracy with diversity)

A limited number of options (users must evaluate each option)

A list with the most option likely first (select it by default)

An easy distinguishing of options (use grouping if too many)

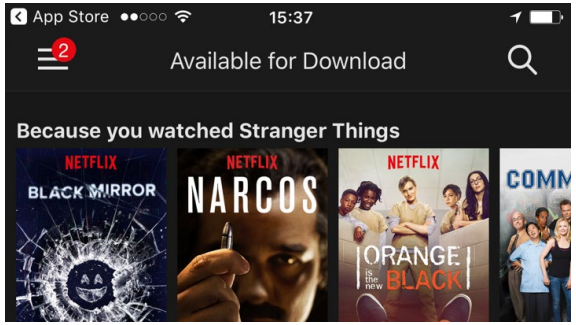


UI/UX Design Principles for Model Output: Confidence

Use confidence numbers from your models to improve UX, but verify that higher confidences lead to higher quality (e.g., review different confidence thresholds).

Know the meaning of your confidence values before using them

Translate confidence value into known concepts, such as attribution „Because you watched X, we suggest … „ or ranking, or numbers for expected use case (e.g., weather forecasts)



Help users in making decisions by translating numbers into suggestions (e.g., recommend waiting to buy an item instead of showing a confidence value to for buying it now)

Adapt the presentation based on confidence values / thresholds (for high, make autonomous actions; for low, ask the user)

For low confidence values and if confidence values match quality, avoid low confidence results appearing in the UI

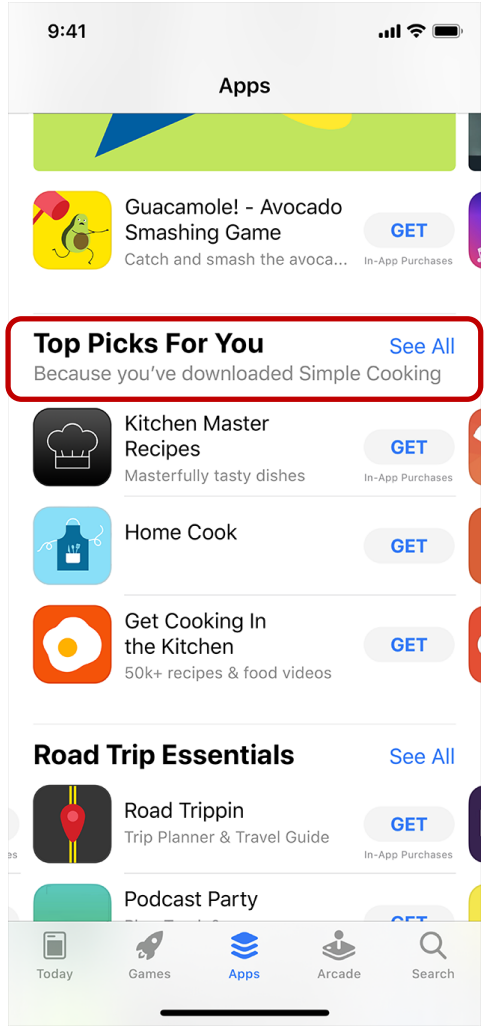
UI/UX Design Principles for Model Output: Attribution

Explains the rationale of an output without too much detail; increases transparency.

Attributions can distinguish multiple options and counteract or fix model mistakes.

Avoid being too specific and too general; try for a balance.

Make and keep attributions factual and on objective features (avoid triggering an emotional reaction, don‘t use features where understanding, beliefs, and people judgement is involved)



UI/UX Design Principles for Model Output: Limitations

Limitations do exist for every AI. We need to set the expectations clear that the limitations are transparent to the users.

Realization:

- Communicate expectations (e.g., describe it in marketing materials)
- Demonstrate app usage, such that users know how to obtain the best result (and what this is)
  Without guidance, people may expect more, for example, use placeholds with keywords (e.g., persons, places) to search for photos; provide feedback to the users on their actions to guide them towards good results; suggest alternative actions instead of presenting no results
- Explain inferior results (requires detecting the cause of a bad result and suggesting the user how to circumvent it, e.g., no photos in the dark)

Sources

Apple UX Design: https://developer.apple.com/design/human-interface-guidelines/machine-learning/
Michael Perlin video lecture on architecture of ML systems https://www.youtube.com/watch?v=zAFFnWh1OsM
https://github.com/joelparkerhenderson/architecture-decision-record
DeepLearning.AI course: ML in Production by Andrew Ng et al.
David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In NeurIPS 2015. 2503–2511.
Hironori Washizaki, Hiromu Uchida, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Studying software engineering patterns for designing machine learning systems. In 10th International Workshop on Empirical Software Engineering in Practice (IWESEP). IEEE, 49–495. https://arxiv.org/pdf/1910.04736.pdf