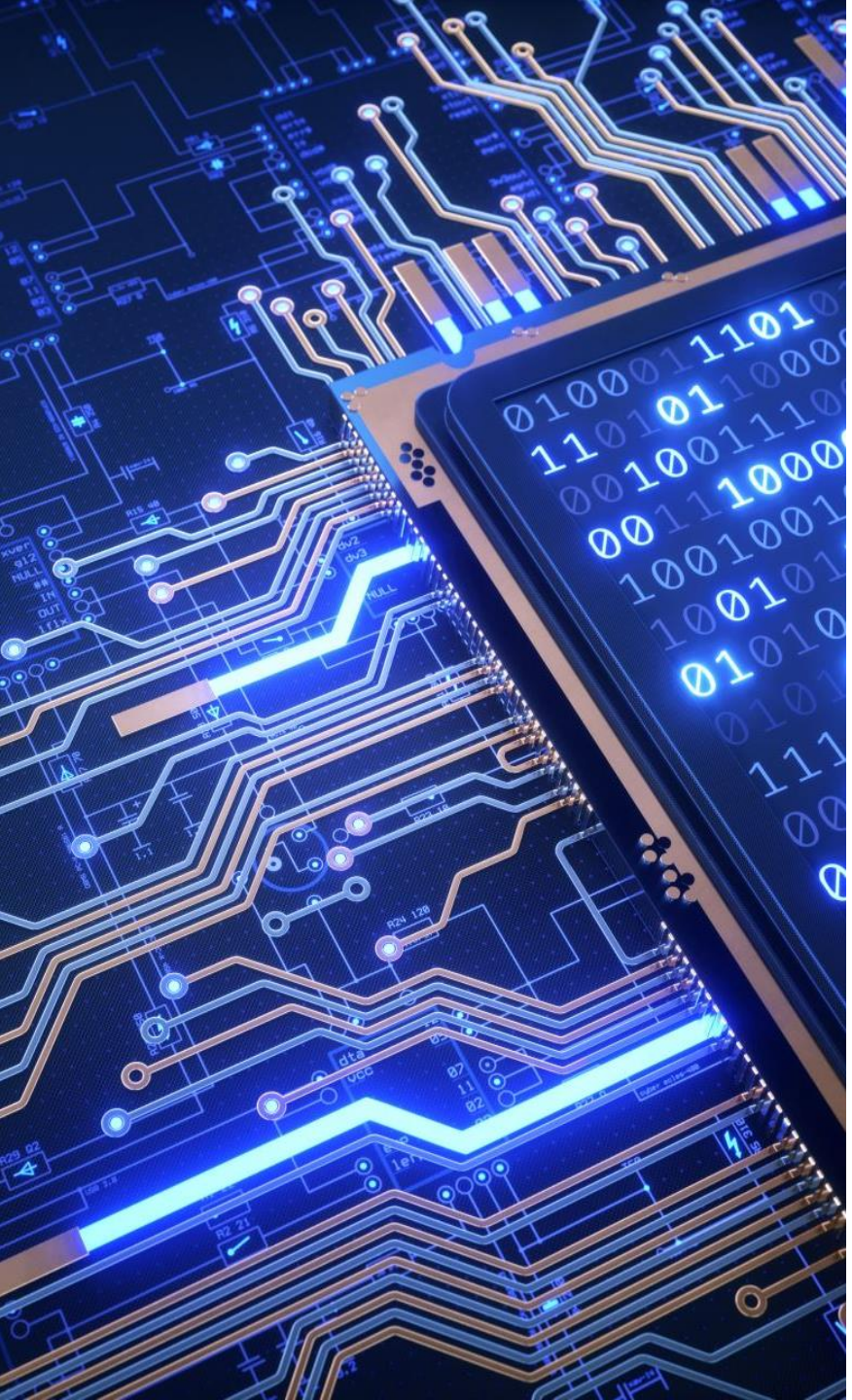


MLOps and beyond

Managing complexity in ML systems by applying engineering principles

Thomas Loeber

Problem: Challenges to Productionizing ML

- Excitement about data science in the 2010s
- But: **Much harder than expected go from ad-hoc Jupyter notebooks to continuously running production software:**
  - Integration with existing IT systems
  - "It works on my machine" problem
  - Performance (Training-serving skew, model drift)
  - Reliability (data issues, infrastructure, code)
  - Maintainability (no observability, tight coupling)
  - Reproducibility (versioning of code is not enough)
  - Compliance
  - Scalability
  - Inefficiency (data scientists spend most of their time cleaning data)

Solution:
Engineering as Managing Complexity

- " Software engineering is programming integrated over time.“

  Saying at Google (according to Titus Winters, Tom Manshrec, & Hyrum Wright: “*Software Engineering at Google*”)

- What exactly is the difference?
  - ➔Need to worry about **maintainability** and **scalability**
- ➔ Engineering is focused on managing **complexity**
  - ➔**Inherent** complexity (unavoidable)
  - ➔**Accidental** complexity (bad)

Examples of *accidental* complexity

- Bad variable naming, bad formatting
- “It works on my machine" problem.
- Time consuming *manual* testing is necessary after every change. Discourages changes.
- Manual steps that need to be executed in just the right order (often combined with localized knowledge)
- No ability to rollback deployment to a previous state (e.g., because some configuration changes happen outside of version control). Applies to both application and infrastructure.
- Bad code: Convoluted logic, nested if-else statements, code is not self-documenting
- Bad design: Tight coupling between components (e.g., overuse of inheritance)
- Lacking observability (e.g., looking through logs is like finding needle in a haystack)
- Handoff points between teams are not formally defined and enforced (e.g., schema definitions for data or APIs; programming interfaces)
- Bad technology choices:
  - Using more powerful technology than necessary for the use case, at the cost of unnecessary engineering overhead. (Most commonly: Kubernetes)
  - Not focusing on the company's core competencies. E.g., a small company without a dedicated infrastructure team maintaining their own servers for observability tools.

-> Much of this applies to both traditional software engineering and ML!

Side note

"Engineering" can also be a

- PR term (Renaming without changing anything, just to gain higher status, is bad)
- aspirational term (Renaming can be a legitimate *part* of a larger strategy of change management)

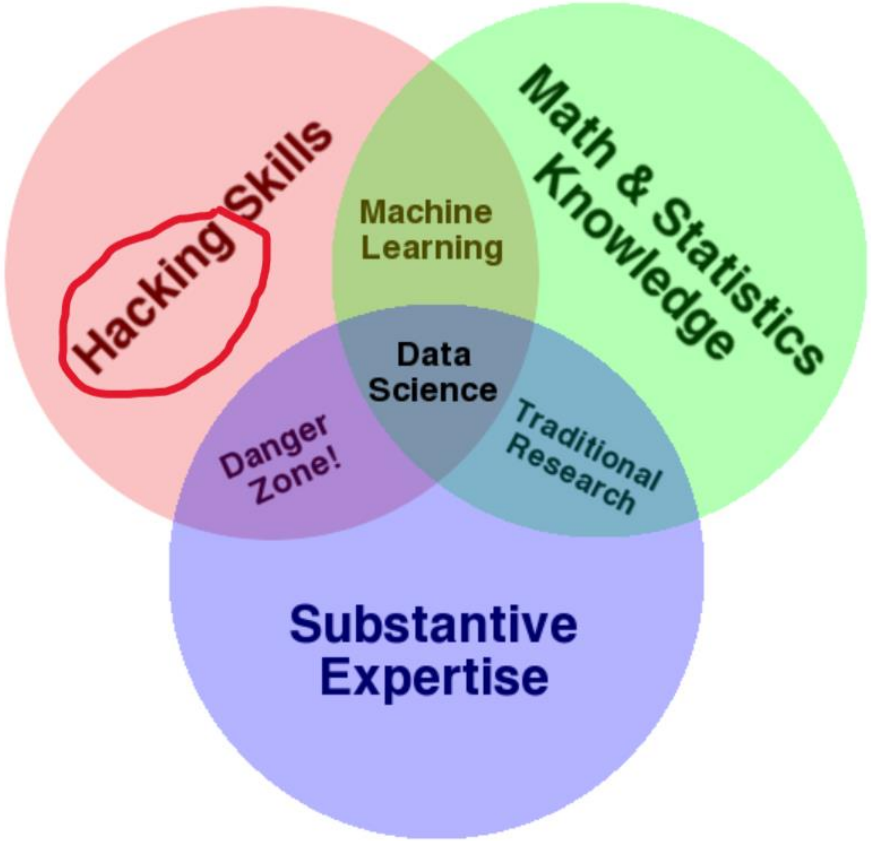
Why Engineering?
Evidence from the *software* world

- Most of the cost of software projects is in **maintenance over time**, not initial creation
  - This becomes even more lopsided with AI-assisted coding
- **Cost of bugs** rises substantially the later in the SDLC they are discovered

  -> Avoiding bugs is much cheaper than fixing them

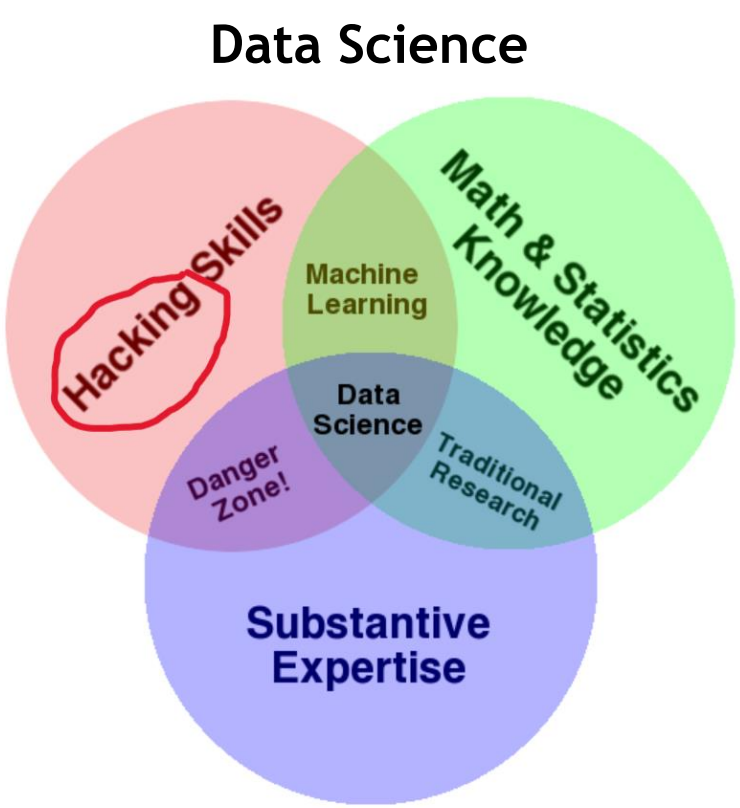
  -> “shifting left”

2010s: Excitement about data science

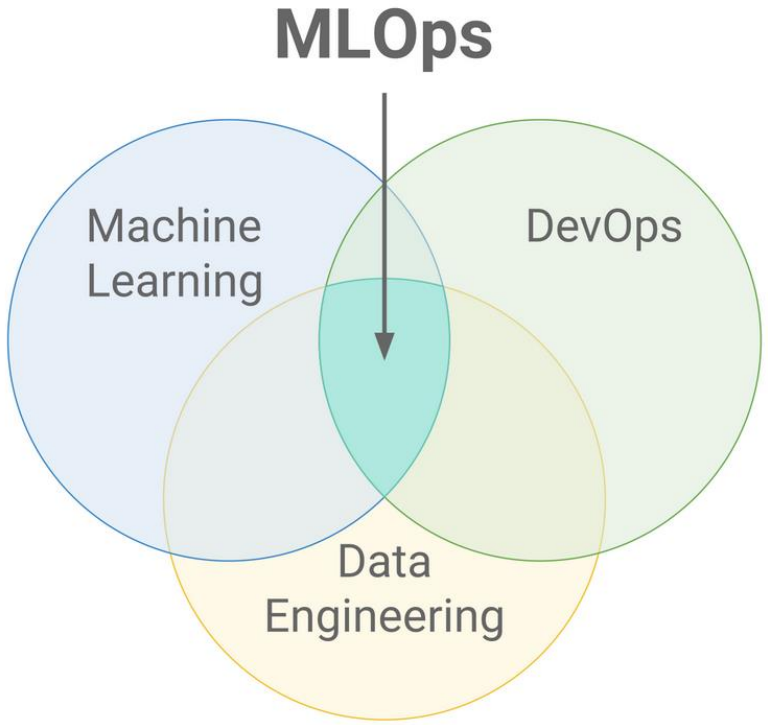


- Drew Conway's "Data Science Venn Diagram“
- One of the most famous early definitions of data science
- http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram

What is MLOps?



http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram



https://en.wikipedia.org/wiki/MLOps#/media/File:ML_Ops_Venn_Diagram.svg

- ➔ Widening the focus beyond model *training* to previously neglected stages in the ML lifecycle
- ➔ **ML**Ops = **Dev**Ops + **Data**Ops + **Model**Ops ?
- ➔ "**Engineering**" rather than "**Hacking**"
- ➔ … but remember there is still a need for both approaches

MLops: Managing the Complexity in ML Systems

- Reproducibility
  - DevOps: Version control of code, Infra-as-Code
  - Additional challenges: Reproducibility requires joint versioning of code, models, and data.
- Data pipelines
  - data engineering: how to run ETL/ELT pipelines reliably (only partially solved), at scale, and with ultra-low latency
  - data lineage/provenance
  - data validation frameworks
  - ML-specific: Feature stores, Data version control
- Deployment
  - DevOps: CI/CD, safe deployment options (gradual, automatic rollback)
  - Canary Deployment -> Champion-Challenger
  - Artifact Repository (binaries, containers) -> Model Repository
- Observability
  - software engineering: logging, monitoring, and distributed tracing (observability of *code* execution)
  - Additionally, in ML we need observability of:
    - Data: data quality, data drift,…
    - Model performance: comparison between different models, comparison of same model over time (model drift?), performance for different subsets of the population (if substantial, does this vary by model?).

➔ **Is MLOps simply the application of engineering principles to ML?**

How?
What we can learn from software engineering

- Clean Code
  - Code is written once but read many times

    -> Code should be optimized for reading (readability), not writing
- Type safety
- Software design patterns
  - Makes code changeable (“Agile”) by decoupling components
  - Most important examples:
    - Program to an interface, not a particular implementation
    - composition over inheritance
- Leverage power of IDEs (even for Jupyter notebooks)
  - Plugins: Static analysis & linting, GitHub Copilot, GitLens
  - Debugger, refactoring engine, auto-formatting
  - better syntax highlighting, go-to-definition/use
- Automated testing
  - Tests are not just there to guarantee quality
  - A trusted test suite also makes sure developers are not afraid to do any necessary *refactoring*
  - Proper acceptance tests serve as documentation of desired behavior (Gherkin)
  - Tests should be *automated* so you don’t have to reinvent the wheel

➔ **Engineering is in part a different *mindset*!**

➔ **We need to codify best practices!**

How? Lessons from software engineering that need to be *adapted*

- Automated testing
  - Challenges:
    - Hard to write unit tests for bad code
    - Many tools are only partially supported in Jupyter notebooks
  - New problems:
    - Static analysis and automatic data validation:
      - Only some of the operations we perform when data frames cannot be cheaply-type checked by static analysis: Column names and types
      - In ML we often want to impose a schema on our data that can only be evaluated at runtime:
        - constraints on a particular column that cannot be represented by built-in datatypes, e.g. a number that must fall in a specific range (such as a probability), uniqueness checks, whether missing values are allowed or not.
        - Constraints on the relationship between columns: one should always be greater than another (e.g., for quantiles); there should be a positive correlation between two columns, etc.
      - Thus, we have to consider the trade-off to what extent the runtime overhead of data validation is worth it. Note, however, that static analysis is even more important for data pipelines than for traditional software engineering because the compute (and time) costs are much higher; therefore, catching typing errors before runtime is paramount.

Challenges:

- *1) Different needs of Data Science and ML Engineering -> need for a hand-off process*
  - *see separate presentation*
- *2) Organizational challenge: Misaligned Incentives (*
  - *see separate presentation*
- *3) Coordination* problem for open-source ML software
  - see below

Challenge 3)
Coordination problem for open-source ML software

- We need to standardize interfaces in order to decouple ML pipelines from specific frameworks used
- Solving this coordination problem is hard because nobody wants to break backwards compatibility
- We could at least develop adapters to translate interfaces – but this may have a negative performance impact
- Unfortunately, the same problem even consists *within* frameworks (e.g., Sagemaker SDK). Or relevant interfaces are not published (e.g., sklearn). This is more easily avoidable.
- This problem is exacerbated by the fact that most of the software packages coming in ML were originally developed by scientists for scientists, and it is not always easy to fix bad design decisions (e.g., overuse of inheritance instead of composition).

Conclusion: Implications for consulting

- Key challenges:
  - Gather ML Engineering best practices
  - Handoff between data scientists and ML engineers
  - Making quality (and technical data) visible through KPIs
  - Implications for client contracts?
    - How do we avoid being blamed down the line for problems caused by short-sighted decisions?
    - Which commitments should we secure from client beforehand to avoid stymieing ML due to dependencies on other teams (e.g., data quality)

- Realistically, clients will vary in their willingness to invest into engineering excellence
  -> Spectrum of high-value consulting to low-cost staffing/outsourcing