Common Pitfalls of DS Projects : Avoiding Common Risks, https://www.youtube.com/watch?v=0GKcsK-oCIA

UNIT IIII

MDSS Chapter 9 Common Pitfalls of DS Projects Session 1

Avoiding Common Risks of DS projects

Objective

We will explore the common pitfalls as well as the mistakes that increase the risks your DS projects and that are easy to commit.

It's important to know how to deal with them for success of your projects.

Different types of DS solutions have many tempting ways of executing the project that can lead to undesired difficulties in the later stages of the project. We will pick and mitigate those issues one by one while following the data science project life cycle.

Topics:

- Avoiding common risks of DS projects
- Approaching research projects
- Dealing with prototypes and **minimum viable product** (**MVP**) projects
- Mitigating risks in production-oriented data science systems

Avoiding Common Risks of DS Projects

The first and most important risk of any data science project is the goal definition. The correct goal definition plays a major part in the success formula. It is often tempting to jump into the implementation stage of the project right after you have the task definition, regardless of whether it is vague or unclear. *By doing this, you risk solving the task in an entirely different way from what the business actually needs*. It is important that you define a concrete and measurable goal that will give your team a tool that they can use to distinguish between right and wrong solutions.

To make sure that the project goal is defined correctly, you may use the following checklist:

- You have a quantifiable business metric that can be calculated from the input data and the algorithm's output.
- The business understands the most important technical metrics that you use.
- The task is defined in DS terminology. You know if you are solving classification, regression, or another kind of task and have an idea of how to approach it technically.
- You understand the details of the (business) process and domain of the problem.

All of the data sources that are required for the start of the project exist and can be accessed by the implementation team.

Another important issue is the documentation. In data science projects, experiment results can often change the course of the project. It is important to log all the experiments, along with the decisions and conclusions that you make. Fixing any incoming changes when it comes to data and requirements is also necessary. When you have this information, you will be able to see the whole line of thought that makes your solution work as it does.

Being aware of the common risks of data science projects will help you recover from major mistakes, but the devil is in the detail.

Approaching Research Projects

A research project is any project that gives you ***solutions to new, not well-known problems.***

Research projects aren't always about advancing science. If your team deals with a new kind of business domain, or a new type of machine learning library, these are also considered to be research projects. Discovering ways to apply data science to new business domains is also research. Almost every data science project includes a research subproject that takes care of the modeling process.

The first pitfall of research projects is the absence of scope. Every research project must have a clear scope, otherwise it won't be possible to finish it. It is also important to fix any external constraints for a research project. Your research budgets will grow as the scope's size increases, so limited budgets may also affect the depth and length of research that you will be able to do.

When you have an idea of your research capacity, you can start filling in the experiment backlog with your team. Each entry in a backlog should contain an idea that may advance your model's quality or help you reach your desired functionality. Each experiment should be defined in accordance with the SMART criteria.

For example, an initial experiment backlog for a binary classification problem may look like this:

1. Perform **exploratory data analysis** (**EDA**) and familiarize yourself with data.

2. Create a baseline model using gradient boosting and basic preprocessing.

3. Test the encoding for categorical variables: Hash encoding.

4. Test the encoding for categorical variables: Target encoding.

5. Feature engineering: Measure the effect of date features.

6. Feature engineering: Aggregate the features day windows.

End of session 1

Dealing with Experiments and prototypes for Risk Aversion ,
https://www.youtube.com/watch?v=6ZYYf4j8uM4

Dealing with Experiments and prototypes for Risk Aversion

It is also important to keep track of everything in an experiment, such as the following:

- Input data
- Date and time of the experiment
- Exact code version that produced the experiment's results
- Output files
- Model parameters and model type
- Metrics

You can use a simple shared document for this purpose or invest in integrating a specialized experiment tracking framework with a user interface (UI) into your project.

Experiment tracking frameworks pay for themselves in projects that have lots of experimentation.

Finally, care about making each experiment reproducible and document the conclusions you drew from the results. To check for reproducibility, use the following criteria for each individual experiment:

- Input data is easily accessible and can be discovered by anyone on the team.
- The experiment code can be run on input data without errors.
- You don't need to enter undocumented configuration parameters to run the experiment. All of the configuration variables are fixed in an experiment configuration.
- The experiment code contains documentation and is easily readable.
- The experiment output is consistent.
- The experiment output contains metrics that you can use for comparison with other experiments.
- Conclusions from experiment results are present in the documentation, comments, or output files.

To avoid pitfalls in research projects, make sure you do the following:

- Define a clear goal
- Define the success criteria
- Define constraints, including the time and budget limitations
- Fill in the experiment backlog

- Prioritize by expectations
- Track all the experiments and their data
- Make the code reproducible
- Document your findings

Successful research projects can help you find an answer to a complex question or push the boundaries of scientific fields and practical disciplines. Research projects are often confused with prototypes, though prototypes pursuit different goals in general.

Dealing with prototypes and MVP projects

If you are dealing with data science, I bet you will find yourself doing a lot of prototyping. Prototypes often have very strict time and money limitations. The first lesson of prototyping is to approach every prototype as an MVP. The key idea behind MVP is to have just enough core features to show a working solution. Bells and whistles can be implemented later, as long as you are able to demonstrate the main idea behind your prototype.

Focusing on core features does not mean that your prototype should not have a pretty UI or stunning data visualizations. If those are the main strengths of your future product, by no means include them. To identify the core features of your product, you should think in terms of markets and processes.

Ask yourself the following questions to check whether a particular feature should be included in the MVP:

- Who are your users?
- What (business) processes are your solutions targeting?
- What problem do you want to solve?
-

Then, look at what features are crucial for reaching the desired goal, what is complementary, and what is unrelated to the goal. After that, look at your competitors and think about what will distinguish your MVP from them. If you have no distinguishing features or you don't solve a different problem, you will have a hard time competing with an already developed and possibly highly adopted competitive product.

Case study – creating an MVP in a consulting company

Our friend Mark works at a consulting company. His team has created a defect detection system prototype for a large manufacturing company. The system should analyze the video stream of products on a conveyor belt and detect products that are defective. Mark has answered the initial MVP question list:

- Who are your users?
  The manufacturing plant product quality department.
- What (business) processes are your solutions targeting?
  We target the core product quality control process.
- What problem do you want to solve?
  We want to decrease the total amount of defected products that are left undetected by the current quality control process.

Using these answers, Mark has created the core feature list for the MVP:

- Defect detection model
- Integration by monitoring cameras over the conveyor belt

Mark knows that the manufacturing plan director, Anthony, who is the key decision-maker, appreciates systems with a slick and intuitive UI. Also, Mark is sure that preparing a model quality report is essential for comparing the efficiency of as-is and to-be quality control processes.

These two insights added some more deliverables to the MVP:

- A UI for monitoring defects in real time
- A model quality report that provides an efficiency comparison between the old process and the improved process, along with an automated quality control step

After the customer approved the scope, Mark decided to use Scrum as a management methodology and focused on delivering the first working version as fast as possible. To get up to speed, Mark's team, who were already experienced in applying computer vision algorithms, used internal software libraries that they can develop for rapid prototyping. The report template meant that they didn't have to spend a lot of time writing documents, which meant they could focus on MVP development.

This example concludes our overview of prototypes. Now, we are ready to dive into the topic of assessing and mitigating risks in data science projects.

Mitigating risks in production-oriented data science systems

End-to-end data science projects encompass one or several full iterations of the data science project life cycle. End-to-end data science projects comprise of all the risks of research projects and MVPs, along with a new set of risks related to change management and production deployment.

The first major risk is the inability to sustain a constant change stream. Data science projects involve scope changes, and you should be able to work with them without making the project fall apart. Scrum gives you the basic tools you need for change management by freezing the development scope over the course of the week. However, for any tool to work, your customer should understand and follow the required processes, along with your team.

Another issue is that the implementation of a given change may cause a lot of unexpected bugs. Data science projects often lack automated testing procedures. The absence of constantly testing existing functionality may cause ripple effects when one simple change causes several bugs. Without tests, a lot more bugs also go unnoticed and get passed into production. It is also important to implement online testing modules since quality assurance does not end in the development phase. Models can degrade in performance over time, and your system should monitor abrupt changes in business and technical metrics.

If your team has not planned for production in advance, you will face many complex engineering issues related to non-functional requirements such as the availability, scalability, and reliability of your system. To avoid this, care about system design and software architecture from the start of the project.

Even if everything has gone well technically, the final result may bewilder your customers. If key stakeholders do not see the benefit of your system, you must look for any mistakes that were in your goal definitions. It is often the case that project goals change midterm, along with the customer's views of what's best for their business. To avoid this major risk, you should constantly check that the task you are working on is important and that the solution method is correct.

In the following table, we have enumerated the common risks and their solutions.

Risk group	Risk	Solution
Common	Vague goal definition.	Make sure that the goal's definition is complete and includes all the items from the checklist in this chapter.
Common	The project goal is not quantifiable.	Define quantifiable business metrics that can be understood by the customer. Define one or several technical metrics that correlate with your business metrics.
Common	Decision-making without keeping track of the record.	Document every major decision and conclusion you make throughout the project. Fix data and code versions in order to reproduce the results that lead to your decisions.
Research	The team can't reproduce the experiment's results.	Track the experiment's results and data, along with the code.
Research	The research has no scope and plan of action.	Plan ahead using the research backlog. Prioritize entries in the research backlog and periodically check whether there are any obsolete entries that should be removed. Assess your expectations of each experiment by performing quick tests, if possible.
MVP	The prototype does not show how to solve the user's problem.	Think about every prototype as an MVP that solves your customers' problems. Define your scope by taking the minimum amount of functionality required to solve your customers' problems into account.
MVP	The MVP includes too many unnecessary features that take time to develop.	Use feature analysis to define the MVP scope.
MVP	The MVP takes a lot of time to develop.	If your team makes a lot of MVPs, think about creating rapid prototyping frameworks and project templates to speed up the development process.

Project development	The customer is constantly pushing the team to make urgent scope changes.	Advocate for the use of Agile development methodology for your project. Track project scope changes to show how they affect project deadlines.
Project development	The customer does not see how the system solves their problem.	Constantly review your project goals and make sure that your way of solving the problem has been confirmed by the customer.
Project development	New changes introduce a lot of bugs.	Write automated tests.
Production deployment	The model's quality is degraded in production and the system has no tools to solve this problem.	Develop an online testing module to track metrics in production. Validate incoming data. Periodically retrain your models on new data.

Production deployment	The system is not suitable for production usage.	Fix functional and nonfunctional requirements for your system. Prepare an architecture vision that provides a production-ready system design.

End of session 2

Thinking of Projects as Products ,
https://www.youtube.com/watch?v=Kn_M21RZ0pc

UNIT IIII

MDSS : Chapter 10 **Creating Products and Improving Reusability** Session 3

Thinking of projects as products

Objective

This chapter will conclude the *Managing Data Science Projects* section by looking at how we can help products grow and improve reusability. This chapter focuses on teams who work to deliver custom solutions for customers. You will find the content in this chapter helpful if your team helps internal business stakeholders at the company or external customers who want to buy expertise, services, and software solutions. The benefits of product thinking and reusability are underappreciated when it comes to the consultation stage, whereas they become more obvious if your team is developing a product that is focused on a market niche.

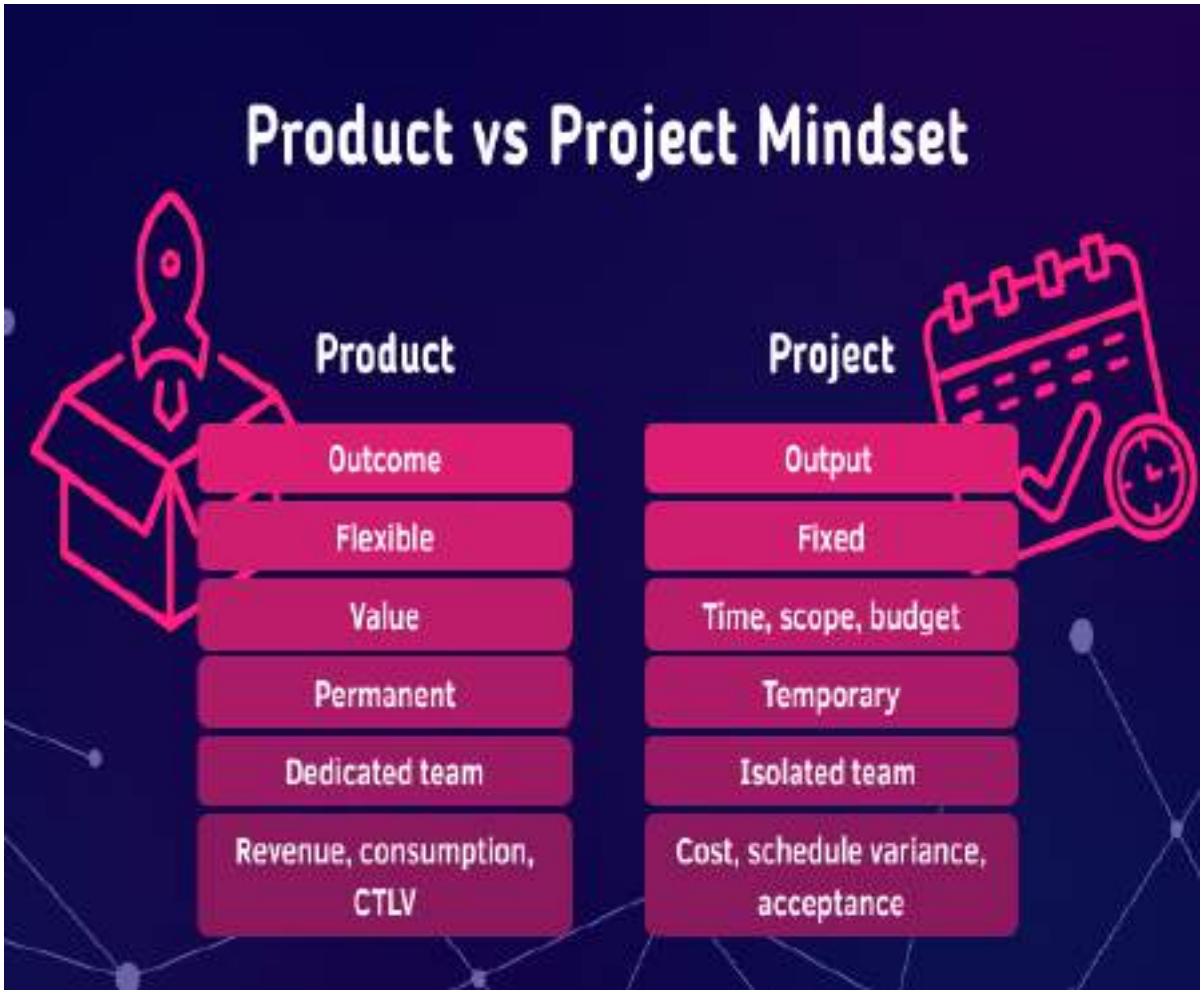
Topics:

- **Thinking of projects as products**
- **Determining the stage of your project**
- **Improving reusability**
- **Seeking and building products**

Thinking of projects as products

Ideas : think about your work as product development.

While many companies deliver software products to the market, you deliver services. We can perceive services as a product; they also obey the laws of supply and demand, and there are markets for different types of services.



Like software products, you can decompose your activity into service features. Some of the aspects of your service that your team is good at will separate your department or company from your competitors, but you will likely find some of these aspects lagging behind other organizations that focus on this particular type of service. For example, a data science consulting company may shine in creating custom models. However, their user interfaces (UIs) will be worse than that of a specialized UI development company. These tradeoffs are justified by market demands: companies who want to buy services on a custom model development rarely need the same team to deliver a high-end UI.

Thinking about services as products opens up new possibilities for improvement. You can analyze your best service feature and think about the improvements that you can introduce to make your service better and more profitable, thereby opening up new possibilities for continuous service quality improvement.

In most cases, the best team delivers more functionality in less time and at a lower cost while being more customizable and better in quality than products that are optimized toward a broader market. As a service provider, you can do one thing that no product is capable of creating solutions that are highly specialized for solving a specific problem for your customer.

The question is, how can we make custom solutions easier to implement while not losing the key benefit of providing a custom solution to a customer's problem?

The first step of thinking about projects in terms of products is to consider the project's stage. In the next section, we will consider the concept of reusability in relation to different project types and stages.

Determining the stage of your project

To develop the action plan of service improvement, you need to determine the type and stage of the project you are working on. We can divide projects into two major categories:

- Products
- Custom solutions

While products are reusable by nature, often, custom solutions aren't. However, custom solutions can be built from reusable components while not losing the qualities of made-to order software. To grow these internal components and improve reusability, you should care about them through each stage of the project:

- **Minimum Viable Product (MVP**): Think about the results of your previous projects that you can reuse while having a minimal investment of time. Even if it looks like building this functionality from scratch will be easier, creating a reusable component can save you a lot more time over a longer time span.
- **Development**: Think about what reusable components you can include in the system that you are building. Decide if any new functionality that you are building can be transformed into a reusable component.
- **Production**: Stabilize and integrate any existing technology.

The next big step in improving reusability is managing research. The question that arises is:

how do we decide when a project needs a research phase?

First, let's look at what kinds of research we can do in data science projects:

Data: Improving your understanding of customer's data by performing

EDA and searching for insights in data.

Models and algorithms: Improving your models and searching for new approaches to solving a problem.

Technology: Researching new technologies to improve your solution. Technology can be directly applicable to your project or supplement the development process by improving operations, requirement management, code or data versioning, and so on.

Research is hard to plan and estimate, but it is necessary for finishing data science projects successfully. It opens up new opportunities, provides insights, and changes processes.

Finding the right balance between research and implementation is crucial for the project's success. To manage research efficiently, you can split your project into two logical subprojects: research and solution development.

You can integrate research phases into the solution development project in several ways, as follows:

In parallel with solution development: This approach requires a separate research team working full time. It is useful because the research process provides results in the background of the main project. Items from the research backlog that yielded good results are converted into research integration tasks in the main project backlog. This way of managing research is useful for data and model research because those activities can take a long time to finish and are straightforward to integrate into the main code base most of the time.

Before or after each development iteration: This approach requires your team to switch focus from solution development to research and is most effective when the results of the research can influence the long-term method of building the system. Technology research is the best candidate for this form of integration.

You can tie technology research into software design stages so that your team can integrate new technologies into a project in a controlled manner.

Improving reusability

Improving reusability is a custom project development setting where you develop and reuse internal components to build better solutions faster. Look at what parts of your work are repeated in all of your projects. For many companies, it's the model deployment and serving. For others, it is building the dashboards on top of the model.

Use open source projects as a starting point. In many fields, the best tools are provided by commercial companies. Thankfully, the data science community is a very open-minded group, and the best tools you can find are open source. Of course, there are great commercial products too, but you can build production-grade systems with state-of-the-art models using open solutions. They will give you a very solid foundation to build upon.

Seeking and building products

Over time, it is natural to have teams that are dedicated to the development of the most complex and popular parts of your reusable tool chain. If you have a reusable component that has stood the test of time in multiple projects and is backed up by an experienced team, it is worth considering turning it into a product.

Another sign that you should make products out of your projects is a high demand for some type of custom solution that you built. If every client asks you to build a customer support chatbot and you have already built tens of them, why not make a product to make everyone’s lives easier?

If your company has no prior experience of delivering new products to the market, take this task seriously and be ready to transition into a new world.

If you want to avoid mistakes, take the time to read the literature related to product management and consider hiring or consulting product managers that have the relevant experience. Product development differs from making a custom solution. In fact, it is an entirely different business, with processes and values that may seem bizarre to a consulting project manager.

Finally, remember that building products is risky and expensive. The product team will need to spend a substantial amount of effort converting any internal code base into a market-ready product. It the idea has no real market demand, this job will be a wasted effort.

Good product ideas are very attractive from a business perspective. However, if you provide services for different customers, you should always think about privacy before turning your projects into products for wide markets.

Privacy Concerns

Privacy is of paramount importance for every data science team, especially for teams that are trusted by other businesses. When you're considering building a new product based on your internal code base or data, always check that everything you plan to use can be sold to other customers. If this is not the case, you may end up building the product from scratch, which will greatly increase the development costs. Ask your company's legal team if all the related NDAs and contracts allow for planned use cases. You also need to consider licensing your internal reusable components so that you can legally reuse them across different projects.

End of session 3

Creating a Development Infrastructure - Implementing ModelOps, https://www.youtube.com/watch?v=Fhh-kG7gv_k

UNIT III

MDSS : Chapter 11 **Creating a Development Infrastructure** Session 4

Implementing ModelOps

Objective

We will look at **ModelOps** and its closest cousin—**DevOps**. We will explore how to build development pipelines for data science and make projects reliable, experiments reproducible, and deployments fast. To do this, we will familiarize ourselves with the general model training pipeline, and see how data science projects differ from software projects from the development infrastructure perspective. We will see what tools can help to version data, track experiments, automate testing, and manage Python environments. Using these tools, you will be able to create a complete **ModelOps** pipeline, which will automate the delivery of new model versions, while taking care of reproducibility and code quality.

In this chapter, we will cover the following:

Topics:

- **Understanding & Looking into ModelOps**
- **Managing code versions and quality**
- **Storing data along with code**
- **Managing environments**
- **Tracking experiments**
- **The importance of automated testing**
- **Continuous model training**
- **A power pack for your projects**

Understanding ModelOps

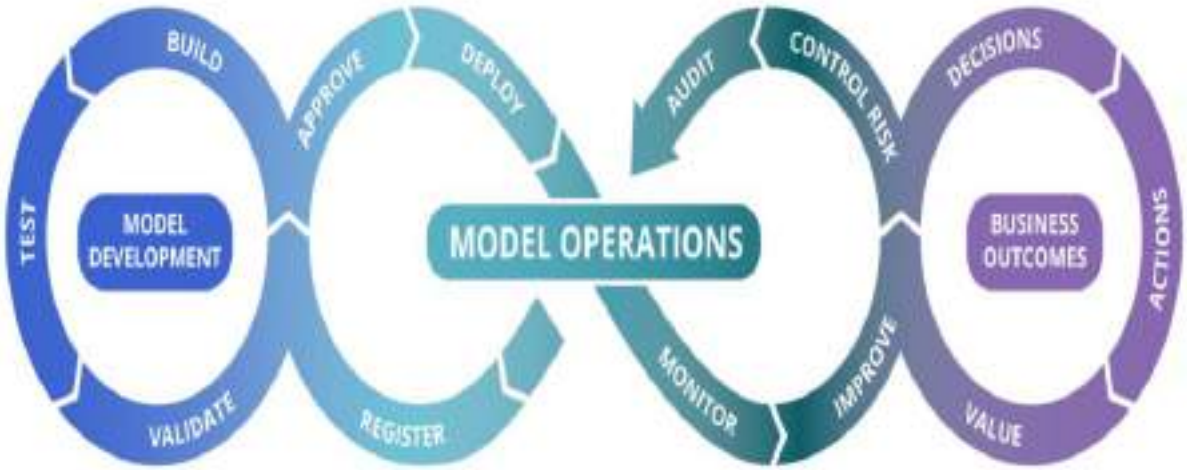
ModelOps is a set of practices for automating a common set of operations that arise in data science projects, which include the following:

Model training pipeline **Data management** **Version control**

Experiment tracking **Testing** **Deployment**

Without **ModelOps**, teams are forced to waste time on those repetitive tasks. Each task in itself is fairly easy to handle, but a project can suffer from mistakes in those steps.

ModelOps helps us to create project delivery pipelines that work like a precise conveyor belt with automated testing procedures that try to catch coding errors.



Looking into DevOps

DevOps stands for development operations. SW development processes include many repetitive and error-prone tasks that should be performed each time software makes a journey from the source code to a working product.

Let's examine a set of activities that comprise the software development pipeline:

1. Performing checks for errors, typos, bad coding habits, and formatting mistakes.
2. Building the code for one or several target platforms. Many applications should work on different operating systems.
3. Running a set of tests that check that the code works as intended, according to the requirements.

4. Packaging the code.

5. Deploying packaged software.

Continuous integration and continuous deployment (CI/CD) states that all of those steps can and should be automated and run as frequently as possible. Smaller updates that are thoroughly tested are more reliable. And if everything goes wrong, it is much easier to revert such an update. Before CI/CD, the throughput of software engineers who manually executed software delivery pipelines limited the deployment cycle speed.
Now, highly customizable CI/CD servers rid us of manual labor, and completely automate all necessary activities. They run on top of a source code version control system, and monitor for new code changes. Once a new code change is present, a CI/CD server can launch the delivery pipeline. To implement DevOps, you need to spend time writing automated tests and defining software pipelines, but after that, the pipeline just works, every time you need it.

DevOps took the software development world by storm, producing many technologies that make software engineers more productive. Like any technology ecosystem, an expert needs to devote time to learning and integrating all tools together. Over time, CI/CD servers became more complicated and feature-rich, and many companies felt the need to have a full-time expert capable of managing delivery pipelines for their projects. Thus, they came up with the role of DevOps engineer.

Many tools from the DevOps world are becoming much easier to use, requiring only a couple of clicks in a user interface. Some CI/CD solutions such as GitLab aid you in creating simple CI/CD pipelines automatically.

Many benefits of CI/CD infrastructure apply to data science projects; however, many areas

remain uncovered. In the next sections of this chapter, we will look at how data science projects can use CI/CD infrastructure, and what tools you can use to make the automation of data science project delivery more complete.

Exploring Special Needs of DS Project Infrastructure

A modern sw project will likely use the following infrastructure to implement CI/CD:

- Version control—Git
- Code collaboration platform—**GitHub, GitLab**
- Automated testing framework—dependent on the implementation language
- CI/CD server—**Jenkins, Travis CI, Circle CI**, or **GitLab CI**

All of these technologies miss several core features that are critical for data science projects:

Data management—tools for solving the issue of storing and versioning large amounts of data files

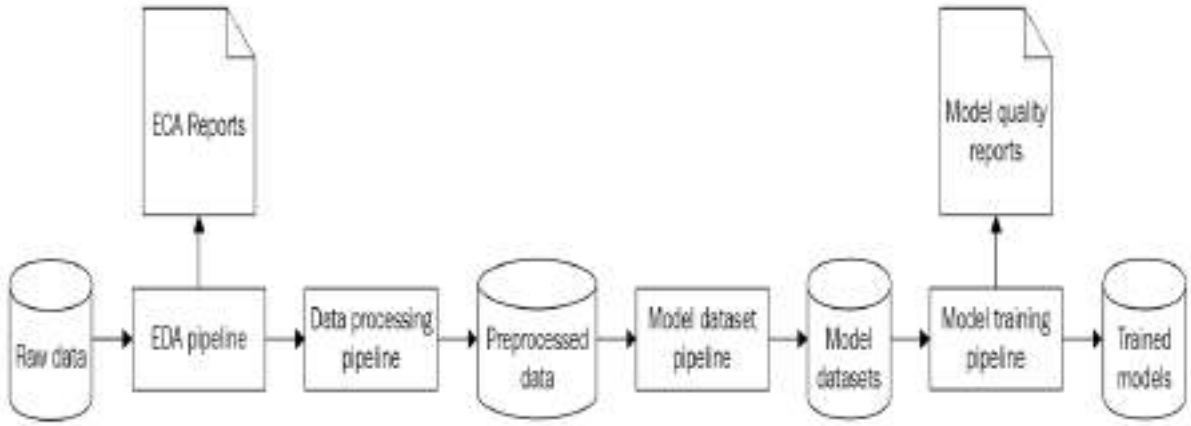
Experiment tracking—tools for tracking experiment results

Automated testing—tools and methods for testing data-heavy applications

Before covering solutions to the preceding issues, we will familiarize ourselves with the data science delivery pipeline.

The DS Delivery Pipeline

Data science projects consist of multiple data processing pipelines that are dependent on each other. The following diagram displays the general pipeline of a data science project:



Stages :

1. Each model pipeline starts with the **Raw data, which is stored in some kind of** data source.

2. Then, data scientists perform **exploratory data analysis (EDA) and create EDA Reports to deepen the understanding of the dataset and discover possible issues** with the data.

4. The **Model dataset pipeline creates ready-to-use datasets for training and testing** models.

5. The **Model training pipeline uses prepared datasets to train models, assess their** quality by performing offline testing, and generate **Model quality reports that** contain detailed information about model testing results.

6. At the end of the pipeline, you get the final artifact—a **Trained model that is** saved on a hard disk or a database.

Implementation strategies and example tools for ModelOps

Managing code versions and quality

DS projects deal with a lot of code, so data scientists need to use source version control (SVC) systems such as Git as a mandatory component. The most obvious way of

using Git is to employ a code collaboration platform such as GitLab or GitHub. Those platforms provide ready-to-use Git servers, along with useful collaboration tools for code reviews and issue management, making working on shared projects easier. Such platforms also offer integrations with CI/CD solutions, creating a complete and easily configurable software delivery pipeline. GitHub and GitLab are free to use, and GitLab is available for on-premises installations, so there is no excuse for your team to miss the benefits of using one of those platforms.

Many teams synonymize Git with one of the popular platform offerings, but it is sometimes useful to know that it is not the only option you have. Sometimes, you have no internet access or the ability to install additional software on server machines but still want the benefits of storing code in a shared repository. You can still use Git in those restricted environments. Git has a useful feature called **file remotes that allow you to push your code** basically everywhere.

For example, you can use a USB stick or a shared folder as a remote repository:

```
git clone --bare /project/location/my-code /remote-location/my-code #copy
your code history from a local git repo
git remote add usb file:///remote/location/my-code
# add your remote as a file location
git remote add usb file:///remote/location/my-code
# add your remote as a file location
git push usb master
# push the code

# Done! Other developers can set up your remote and pull updates:
git remote add usb file:///remote/location/my-code # add your remote as a
file location
git pull usb mater # pull the code
```

By changing the `file:///` path to the `ssh:///` path, you can also push code to the remote SSH machines on your local network.

Most data science projects are written in Python, where static code analysis and code build systems are not as widespread as in other programming languages. Those tools allow you to groom code automatically and check it for critical errors and possible bugs each time you try to build a project. Python has such tools too—look at pre-commit (`https://pre-commit.com`).

The following screenshot demonstrates the running of pre-commit on a Python code repository:

Having covered the main recommendations for handling code, let's now see how we can achieve the same results for data, which is an integral part of any data science project.

Storing data along with the code

As you have seen previously, we can structure code in data science projects into a set of pipelines that produce various artifacts: reports, models, and data. Different versions of code produce changing outputs, and data scientists often need to reproduce results or use artifacts from past versions of pipelines.

This distinguishes data science projects from software projects and creates a need for managing data versions along with the code: **Data Version Control (DVC).** In general, different software versions can be reconstructed by using the source code alone, but for data science projects this is not sufficient. Let's see what problems arise when you try to track datasets using Git.

Tracking and versioning data

To train and switch between every version of your data science pipeline, you should track data changes along with the code. Sometimes, a full project pipeline can take days to calculate. You should store and document not only incoming but also intermediate datasets for your project to save time. It is handy to create several model training pipelines from a single dataset without waiting for the dataset pipeline to finish each time you need it. Structuring pipelines and intermediate results is an interesting topic that deserves special attention. The pipeline structure of your project determines what intermediate results are available for use. Each intermediate result creates a branching point, from where several other pipelines can start. This creates the flexibility of reusing intermediate results, but at the cost of storage and time. Projects with lots of intermediate steps can consume a lot of disk space and will take more time to calculate, as disk input/output takes a lot of time.

Be aware that model training pipelines and production pipelines should be different. A model training pipeline might have a lot of intermediate steps for research flexibility, but a production pipeline should be highly optimized for performance and reliability. Only intermediate steps that are strictly necessary to execute the finalized production pipeline need to be executed.

Storing data files is necessary for reproducing results but is not sufficient for understanding them. You can save yourself a lot of time by documenting data descriptions, along with all reports that contain summaries and conclusions that your team draws from data. If you can, store those documents in a simple textual format so that they can be easily tracked in your version control system along with the corresponding code.

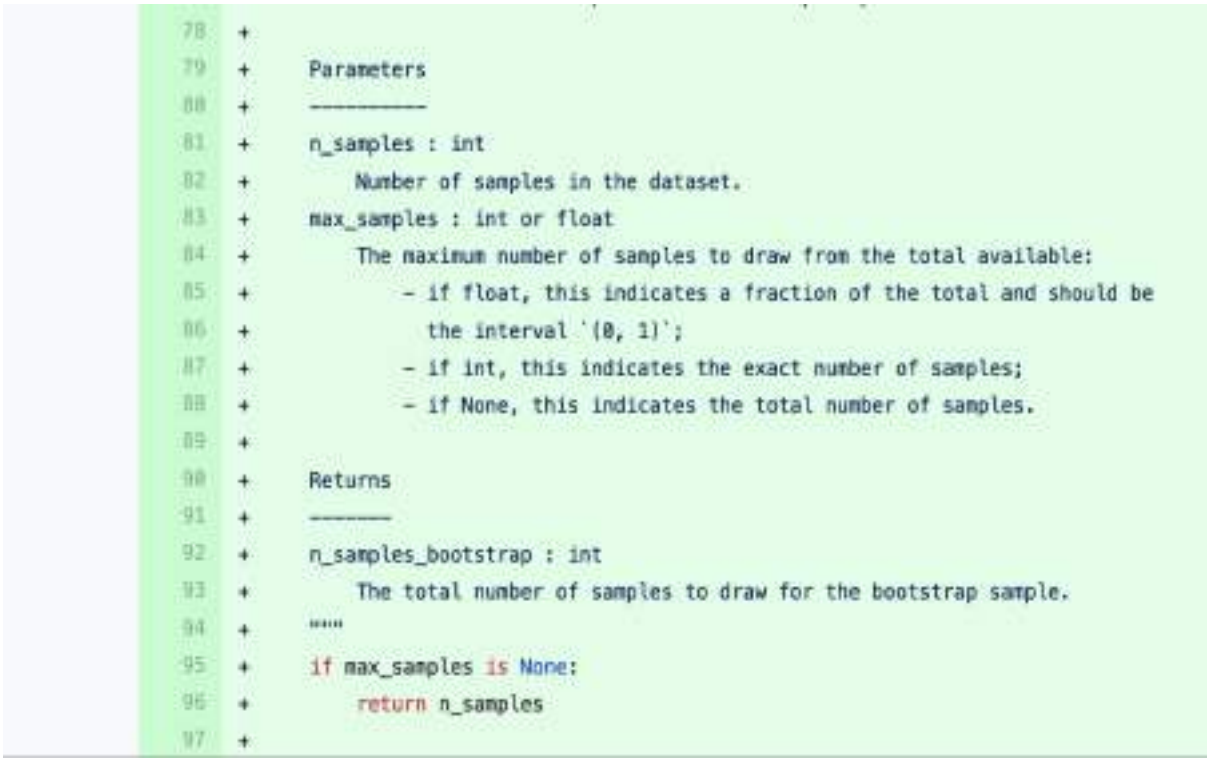
You can use the following folder structure to store the data in your projects:

- Project root:
  - Data:
    - Raw—raw data from your customer
    - Interim—intermediate data generated by the processing pipeline
    - Preprocessed—model datasets or output files
  - Reports—project reports for EDA, model quality, and so on
  - References—data dictionaries and data source documentation

Storing data in practice

We have explored why it is important to store and manage data artifacts along with the code but did not look at how we can do it in practice. Code version control systems such as Git are ill-suited for this use case. Git was developed specifically for storing source code changes. Internally, each change in Git is stored as a diff file that represents changed lines of a source code file.

You can see a simple example of a diff file in the following screenshot:



```
+
+    Parameters
+    ----------
+    n_samples : int
+        Number of samples in the dataset.
+    max_samples : int or float
+        The maximum number of samples to draw from the total available:
+            - if float, this indicates a fraction of the total and should be
+              the interval `(0, 1)`;
+            - if int, this indicates the exact number of samples;
+            - if None, this indicates the total number of samples.
+
+    Returns
+    -------
+    n_samples_bootstrap : int
+        The total number of samples to draw for the bootstrap sample.
+    """
+    if max_samples is None:
+        return n_samples
+
```

The highlighted lines marked with **+** represent added lines,

while highlighted lines marked with **–** stand for deleted lines.

Adding large binary or text files in Git is considered bad practice because it results in massive redundant **diff** computations, which makes repositories slow to work with and large in size

diff files serve a very specific problem: they allow developers to browse, discuss, and

switch between sets of changes. diff is a line-based format that is targeted at text files. On the contrary, small changes in binary data files will result in a completely different data file.

In such cases, Git will generate a massive diff for each small data modification.

In general, you needn't browse or discuss changes to a data file in a line-based format, so calculating and storing diff files for each new data version is unnecessary: it is much

simpler to store the entire data file each time it changes.

End of session 4

Managing Environment and Tracking Experiments & Packaging Code

https://www.youtube.com/watch?v=o5P7QT8FH0w

UNIT III

MDSS : Chapter 11 Session 5

Managing Environment and Tracking Experiments & Packaging Code

Managing Environments

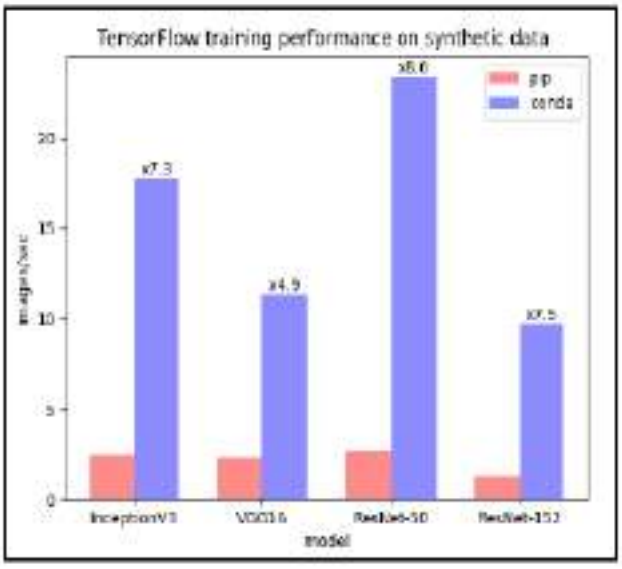
Data science projects depend on a lot of open source libraries and tools for doing data analysis. Many of those tools are constantly updated with new features, which sometimes break APIs. It is important to fix all dependencies in a shareable format that allows every team member to use the same versions and build libraries.

The Python ecosystem has multiple environment management tools that take care of different problems. Tools overlap in their use cases and are often confusing to choose from, so we will cover each briefly:

- **pyenv** (`https://github.com/pyenv/pyenv`) is a tool for managing Python distributions on a single machine. Different projects may use different Python versions, and pyenv allows you to switch between different Python versions between projects.
- **virtualenv** (`https://virtualenv.pypa.io`) is a tool for creating virtual environments that contain different sets of Python packages. Virtual environments are useful for switching contexts between different projects, as they may require the use of conflicting versions of Python packages.

- **pipenv** (`https://pipenv-searchable.readthedocs.io`) is a step above virtualenv. Pipenv cares about automatically creating a sharable virtual environment for a project that other developers may easily use.
- **Conda** (`https://www.anaconda.com/distribution/`) is another environment manager like pipenv. Conda is popular in the data science community for several reasons:
    - It allows sharing environments with other developers via the `environment.yml` file.
    - It provides the Anaconda Python distribution, which contains gigabytes of pre-installed popular data science packages.
    - It provides highly optimized builds of popular data analysis and machine learning libraries. Scientific Python packages often require building dependencies from the source code.
    - Conda can install the CUDA framework along with your favorite deep learning framework. CUDA is a specialized computation library that is required for optimizing deep neural networks on a GPU.

Consider using conda for managing data science project environments if you are not doing so already. It will not only solve your environment management problems but also save time by speeding up the computation. The following plot shows the performance difference between using the TensorFlow libraries installed by pip and conda (you can find the original article by following this link: `https://www.anaconda.com/tensorflow-in-anaconda/`):



Tracking experiments

Experimentation lies at the core of data science. Data scientists perform many experiments to find the best approach to solving the task at hand. In general, experiments exist in sets that are tied to data processing pipeline steps.

For example, your project may comprise the following experiment sets:

- Feature engendering experiments
- Experiments with different machine learning algorithms
- Hyperparameter optimization experiments

Each experiment can affect the results of other experiments, so it is crucial to be able to reproduce each experiment in isolation. It is also important to track all results so your team can compare pipeline variants and choose the best one for your project according to the metric values.

A simple spreadsheet file with links to data files and code versions can be used to track all experiments, but reproducing experiments will require lots of manual work and is not guaranteed to work as expected. Although tracking experiments in a file requires manual work, the approach has its benefits: it is very easy to start and pleasant to version. For example, you can store the experiment results in a simple CSV file, which is versioned in Git along with your code.

A recommended minimum set of columns for a metric tracking file is as follows:

1. Experiment date
2. Code version (Git commit hash)
3. Model name
4. Model parameters
5. Training dataset size

6. Training dataset link
7. Validation dataset size (fold number for cross-validation)
8. Validation dataset link (none for cross-validation)
9. Test dataset size
10. Test dataset link
11. Metric results (one column per metric; one column per dataset)
12. Output file links
13. Experiment description

Files are easy to work with if you have a moderate amount of experiments, but if your project uses multiple models, and each requires a large amount of experimentation, using files becomes cumbersome. If a team of data scientists performs simultaneous experiments, tracking files from each team member will require manual merges, and data scientists are better off spending time on carrying out more experiments rather than merging other teammates' results. Special frameworks for tracking experiment results exist for more complex research projects. These tools integrate into the model training pipeline and allow you to automatically track experiment results in a shared database so that each team member can focus on experimentation, while all bookkeeping happens automatically. Those tools present a rich user interface for searching experiment results, browsing metric plots, and even storing and downloading experiment artifacts. Another benefit of using experiment tracking tools is that they track a lot of technical information that might become handy but is too tedious to collect by hand: server resources, server hostnames, script paths, and even environment variables present on the experiment run.

The data science community uses three major open source solutions that allow the tracking of experiment results. These tools pack much more functionality than experiment tracking, and we will briefly cover each:

Sacred: This is an advanced experiment tracking server with modular architecture. It has a Python framework for managing and tracking experiments that can be easily integrated into the existing code base. Sacred also has several UIs that your team can use to browse experiment results. Out of all the other solutions, only Sacred focuses fully on experiment tracking. It captures the widest set of information, including server information, metrics, artifacts, logs, and even experiment source code. Sacred presents the most complete experiment tracking experience, but is hard to manage, since it requires you to set up a separate tracking server that should always be online. Without access to the tracking server, your team won't be able to track experiment results.

MLflow: This is an experimentation framework that allows tracking experiments, serving models, and managing data science projects. MLflow is easy to integrate and can be used both in a client-server setup or locally. Its tracking features lag a bit behind Sacred's powerhouse but will be sufficient for most data science projects. MLflow also provides tools for jumpstarting projects from templates and serving trained models as APIs, providing a quick way to publish experiment results as a production-ready service.

DVC: This is a toolkit for data versioning and pipeline management. It also provides basic file-based experiment tracking functionality, but it is subpar in terms of usability compared to MLflow and Sacred. The power of DVC lies in experiment management: it allows you to create fully versioned and reproducible model training pipelines. With DVC, each team member is able to pull code, data, and pipelines from a server and reproduce results with a single command. DVC has a rather steep learning curve but is worth learning, as it solves many technical problems that arise in collaboration on data science projects. If your metric tracking requirements are simple, you can rely on DVC's built-in solution, but if you need something more rich and visual, combine DVC with MLflow or Sacred tracking—those tools are not mutually exclusive.

The importance of automated testing

Automated testing is considered to be mandatory in software engineering projects. Slight changes in software code can introduce unintended bugs in other parts, so it is important to check that everything works as intended as frequently as possible. Automated tests that are written in a programming language allow testing the system as many times as you like. The principle of CI advises running tests each time a change in code is pushed to a version control system. A multitude of testing frameworks exists for all major programming languages. Using them, developers can create automated tests for the backend and frontend parts of their product. Large software projects can include thousands of automated tests that are run each time someone changes the code. Tests can consume significant resources and require a lot of time for completion. To solve this problem, CI servers can run tests in parallel on multiple machines.

In software engineering, we can divide all tests into a hierarchy:

1. End-to-end tests perform a full check of a major function of a system. In data science projects, end-to-end tests can train a model on a full dataset and check whether the metrics values suffice minimum model quality requirements.

2. Integration tests check that every component of the system works together as intended. In a data science system, an integration test might check that all of the steps of the model testing pipeline finish successfully and provide the desired result.

3. Unit tests check individual classes and functions. In a data science project, a unit test can check the correctness of a single method in a data processing pipeline step.

If the world of software testing is so technologically developed, can data science projects benefit from automated tests? The main difference between data science code and software testing code is the reliability of data. A fixed set of test data that is generated before a test run is sufficient for most software projects. In data science projects, the situation is different. For a complete test of the

model training pipeline, you may need gigabytes of data. Some pipelines may run for hours or even days and require distributed computation clusters, so testing them becomes impractical. For this reason, many data science projects avoid automated testing. Thus, they suffer from unexpected bugs, ripple effects, and slow change integration cycles.

A ripple effect is a common software engineering problem when a slight change in one part of the system can affect other components in an unexpected way, causing bugs. Automated tests are an efficient solution for detecting ripple effects before they cause any real damage.

Despite the difficulties, the benefits of automated testing are too great to ignore. Ignoring tests turns out to be much costlier than building them. This is true for data science projects and software projects. The benefits of automated testing grow with project size, complexity, and team size. If you lead a complex data science project, consider automating testing as a mandatory requirement for your project.

Let's look at how we can approach testing data science projects. End-to-end testing for model training pipelines might be impractical, but what about testing individual pipeline steps? Apart from the model training code, each data science project will have some business logic code and data processing code. Most of this code can be abstracted away from distributed computation frameworks in isolated classes and functions that are easy to test.

If you architect a project's code base with tests in mind from the start, it will be much easier to automate testing. Software architects and lead engineers on your team should take the testability of the code as one of the main acceptance criteria for code reviews. If the code is properly encapsulated and abstracted, testing becomes easier.

In particular, let's take the model training pipeline. If we separate it into a series of steps

with clearly defined interfaces, we can then test data preprocessing code separately from

model training code. And if data preprocessing takes a lot of time and requires expensive computation resources, you can at least test individual parts of the pipeline. Even basic function-level tests (unit tests) can save you a lot of time, and it is much easier to transition to full end-to-end tests from the basis of unit tests.

To benefit from automated testing in your projects, start from the following guidelines:

- Architect your code for better testability.
- Start small; write unit tests.
- Consider building integration and end-to-end tests.
- Keep at it. Remember that testing saves time—especially those nights when your team has to fix unexpected bugs in freshly deployed code.

We have seen how to manage, test, and maintain code quality in data science projects. Next, let's look at how we can package code for deployment.

Packaging Code

When deploying Python code for data science projects, you have several options:

- **Regular Python scripts:** You just deploy a bunch of Python scripts to the server and run them. This is the simplest form of deployment, but it requires a lot of manual preparation: you need to install all required packages, fill in configuration files, and so on. While those actions can be automated by using tools such as Ansible (https:/ / www. ansible. com/), it's not recommended to use this form of deployment for anything but the simplest projects with no long-term maintainability goals.

Python packages: Creating a Python package using a setup.py file is a much more convenient way to package Python code. Tools such as PyScaffold provide ready-to-use templates for Python packages, so you won't need to spend much time structuring your project. In the case of Python packages, Ansible still remains a viable option for automating manual deployment actions.

Docker image: Docker (https:/ / www. docker. com/) is based on a technology called Linux containers. Docker allows packaging your code into an isolated portable environment that can be easily deployed and scaled on any Linux machine. It's like packaging, shipping, and running your application along with all dependencies, including a Python interpreter, data files, and an OS distribution without entering the world of heavyweight virtual machines. Docker works by building a **Docker** image from a set of commands specified in a **Dockerfile.**

A running instance of a Docker image is called a Docker container.

Now, we are ready to integrate all tools for dealing with code, data, experiments, environments, testing, packaging, and deployment into a single coherent process for delivering machine learning models.

Continuous Model Training

The end goal of applying CI/CD to data science projects is to have a continuous learning pipeline that creates new model versions automatically. This level of automation will allow your team to examine new experiment results right after pushing the changed code. If everything works as expected, automated tests finish, and model quality reports show good results, the model can be deployed into an online testing environment.

Let's describe the steps of continuous model learning:

1. CI:
    1. Perform static code analysis.
    2. Launch automated tests.
2. Continuous model learning:
    1. Fetch new data.
    2. Generate EDA reports.
    3. Launch data quality tests.
    4. Perform data processing and create a training dataset.
    5. Train a new model.

6. Test the model's quality.
7. Fix experiment results in an experiment log.

3. CD:

1. Package the new model version.
2. Package the source code.
3. Publish the model and code to the target server.
4. Launch a new version of the system.

CI/CD servers can automate all parts of the preceding pipeline. CI/CD steps should be easy to handle, as they are what CI/CD servers were created for. Continuous model learning should not be hard either, as long as you structure your pipeline so that it can be launched automatically from the command line. Tools such as DVC can aid you in creating reproducible pipelines, which makes it an attractive solution for the continuous model learning pipeline.

Case study – building ModelOps for a predictive maintenance system

Oliver is a team leader of a DS science project for a large manufacturing company called **MannCo**, whose plants can be found in multiple cities around the country. Oliver's team developed a predictive maintenance model that can help **MannCo** to forecast and prevent expensive equipment breakages, which result in costly repairs and long production line outages. The model takes measurements of multiple sensors as input and outputs a package probability that can be used to plan a diagnostics and repair session.

Each piece of this equipment is unique in its own way because it operates under different conditions on each one of **MannCo**'s plants. This meant that Oliver's team would need to

constantly adapt and retrain separate models for different plants. Let's look at how they solved this task by building a **ModelOps** pipeline.

There were several data scientists on the team, so they needed a tool for sharing the code with each other. The customer requested that, for security purposes, all code should be stored in local company servers, and not in the cloud.

Oliver decided to use GitLab **(https:/ / about. gitlab. com/),** as it was a general practice in the company.

In terms of the overall code management process, Oliver suggested using GitFlow (https:/ / danielkummer. github. io/ git- flow- cheatsheet/).

It provided a common set of rules for creating new features, releases, and hotfixes for every team member.

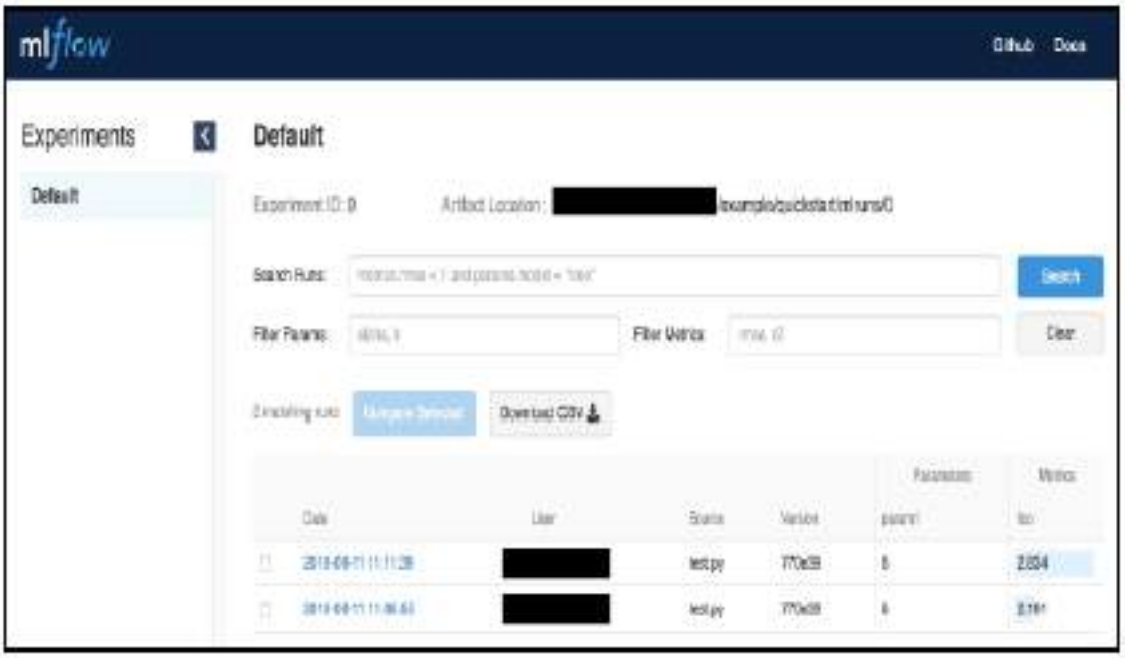
Oliver knew that a reliable project structure would help his team to properly organize code, notebooks, data, and documentation, so he advised his team to use PyScaffold (https:/ / pyscaffold. readthedocs. io/) along with the plugin for data science projects (https:/ / github. com/ pyscaffold/ pyscaffoldext- dsproject). PyScaffold allowed them to bootstrap a project template that ensured a uniform way to store and version data science projects. PyScaffold already provided the environment.yml file, which defined a template Anaconda (https:/ / www. anaconda. com/ distribution/) environment, so the team did not forget to lock the package dependencies in a versioned file from the start of the project. Oliver also decided to use DVC (https:/ / dvc. org/) to version datasets using the company's internal SFTP server. They also used a --gitlab flag for the pyscaffold command so that they would have a ready-to-use GitLab CI/CD template when they needed it.

The project structure looked like this (taken from the `pyscaffold-dsproject` documentation):

```
├── AUTHORS.rst <- List of developers and maintainers.
├── CHANGELOG.rst <- Changelog to keep track of new features and fixes.
├── LICENSE.txt <- License as chosen on the command-line.
├── README.md <- The top-level README for developers.
├── configs <- Directory for configurations of model & application.
├── data
│   ├── external <- Data from third party sources.
│   ├── interim <- Intermediate data that has been transformed.
│   ├── processed <- The final, canonical data sets for modeling.
│   └── raw <- The original, immutable data dump.
├── docs <- Directory for Sphinx documentation in rst or md.
├── environment.yaml <- The conda environment file for reproducibility.
├── models <- Trained and serialized models, model predictions,
│   or model summaries.
├── notebooks <- Jupyter notebooks. Naming convention is a number (for
│   ordering), the creator's initials and a description,
│   e.g. `1.0-fw-initial-data-exploration`.
├── references <- Data dictionaries, manuals, and all other materials.
├── reports <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures <- Generated plots and figures for reports.
├── scripts <- Analysis and production scripts which import the
```

```
|  actual PYTHON_PKG, e.g. train_model.
├── setup.cfg <- Declarative configuration of your project.
├── setup.py <- Use `python setup.py develop` to install for development
or
|  or create a distribution with `python setup.py bdist_wheel`.
├── src
|  └── PYTHON_PKG <- Actual Python package where the main functionality
goes.
├── tests <- Unit tests which can be run with `py.test`.
├── .coveragerc <- Configuration for coverage reports of unit tests.
├── .isort.cfg <- Configuration for git hook that sorts imports.
└── .pre-commit-config.yaml <- Configuration of pre-commit git hooks.
```

The project team quickly discovered that they would need to perform and compare many experiments to build models for different manufacturing plants. They evaluated DVC's metric tracking capabilities. It allowed tracking all metrics using a simple versioned text file in Git. While the feature was convenient for simple projects, it would be hard to use it in a project with multiple datasets and models. In the end, they decided to use a more advanced metric tracker—MLflow (`https://mlflow.org`). It provided a convenient UI for browsing experiment results and allowed using a shared database so that every team member would be able to quickly share their results with the team. MLflow was installed and configured as a regular Python package, so it easily integrated into the existing technology stack of the project:



The team also decided to leverage DVC pipelines to make each experiment easily reproducible. The team liked to prototype models using Jupyter notebooks, so they decided to use papermill (`https://papermill.readthedocs.io/en/latest/`) to work with notebooks as they were a set of parametrized Python scripts. Papermill allows executing Jupyter notebooks from the command line without starting Jupyter's web interface. The team found the functionality very convenient to use along with the DVC pipelines, but the command line for running a single notebook started to be too long:

```
dvc run -d ../data/interim/ -o ../data/interim/01_generate_dataset -o
../reports/01_generate_dataset.ipynb papermill --progress-bar --log-output
--cwd ../notebooks ../notebooks/01_generate_dataset.ipynb
../reports/01_generate_dataset.ipynb
```

To solve this problem, they wrote a Bash script to integrate DVC with papermill so that the team members could create reproducible experiments with less typing in the terminal:

```
#!/bin/bash
set -eu
```

To solve this problem, they wrote a Bash script to integrate DVC with papermill so that the team members could create reproducible experiments with less typing in the terminal:

```
#!/bin/bash
set -eu

if [ $# -eq 0 ]; then
 echo "Use:"
 echo "./dvc-run-notebook [data subdirectory] [notebook name] -d [your DVC
dependencies]"
 echo "This script executes DVC on a notebook using papermill. Before
running create ../data/[data subdirectory] if it does not exist and do not
forget to specify your dependencies as multiple last arguments"
 echo "Example:"
 echo "./dvc-run-notebook interim ../notebooks/02_generate_dataset.ipynb -d
../data/interim/"
 exit 1
fi

NB_NAME=$(basename -- "$2")

CMD="dvc run ${*:3} -o ../data/$1 -o ../reports/$NB_NAME papermill --
progress-bar --log-output --cwd ../notebooks ../notebooks/$NB_NAME
../reports/$NB_NAME"

echo "Executing the following DVC command:"
echo $CMD
$CMD
```

When using several open source ModelOps tools in a single project, your team might need to spend some time integrating them together. Be prepared, and plan accordingly.

Over time, some parts of the code started to duplicate inside the notebooks. The PyScaffold template provides a way to solve this problem by encapsulating repeated code in the project's package directory—`src`. This way, the project team could quickly share code between notebooks. To install the project's package locally, they simply used the following command from the project's root directory:

```
pip install -e .
```

Closer to the project release date, all stable code bases migrated to the project's `src` and `scripts` directories. The `scripts` directory contained a single entry point script for training a new model version that was output into the `models` directory, which was tracked by DVC.

To be sure that new changes did not break anything important, the team wrote a set of automated tests using `pytest` (`https://docs.pytest.org/`) for the stable code base. The tests also checked model quality on a special test dataset created by the team. Oliver modified a GitLab CI/CD template that was generated by PyScaffold so that tests would be run with each new commit that was pushed in a Git repository.

The customer requested a simple model API, so the team decided to use an MLflow server (`https://mlflow.org/docs/latest/models.html`), as MLflow was already integrated into the project. To further automate the deployment and packaging process, the team decided to use Docker along with GitLab CI/CD. To do this, they followed GitLab's guide for building Docker images (`https://docs.gitlab.com/ee/ci/docker/using_docker_build.html`).

The overall ModelOps process for the project contained the following steps:

1. Create new changes in the code.
2. Run pre-commit tests for code quality and styling (provided by PyScaffold).
3. Run pytest tests in GitLab CI/CD.
4. Package code and trained models into a Docker image in GitLab CI/CD.
5. Push the Docker image into the Docker registry in GitLab CI/CD.
6. After manual confirmation in the GitLab UI, run the `update` command on the customer server. This command simply pushes the new version of the Docker image from the registry to the customer's server and runs it instead of the old version. If you're wondering how you can do this in GitLab CI/CD, take a look here: `https://docs.gitlab.com/ee/ci/environments.html#configuring-manual-deployments`.

Please note that, in real projects, you may want to split the deployment into several stages for at least two different environments: staging and production.

Creating an end-to-end ModelOps pipeline streamlined the deployment process and allowed the team to spot bugs before they went into production so that the team was able to focus on building models instead of carrying out repetitive actions to test and deploy new versions of a model.

As a conclusion to this chapter, we'll look at a list of tools that you can use to build ModelOps pipelines.

End of session 5

Building Your Technology Stack ,
https://www.youtube.com/watch?v=yAGooMIZIM0

UNIT III

MDSS : Chapter 12 **Building Your Technology Stack** Session 6

Objective

Technology choices have lasting consequences. A project's technology stack determines the functional and nonfunctional capabilities of your system, so it is critical to make thoughtful choices. The bidirectional link between technologies and requirements opens up an analytical approach for choosing between different technologies by matching their features against the project's needs. In this chapter, we will see how we can use software design practices to form project-specific technology stacks and see what technologies should constitute the core technology stack that's shared among all of your projects. We will also explore an approach that compares different technologies so that you can make a rational choice between apparently similar options.

Topics:

- **Defining the elements of the technology stack**
- **Choosing between core- and project-specific technologies**
- **Comparing tools and products**

Defining the elements of a technology stack

A technology stack is a set of tools that your team uses to deliver products and finish projects. When choosing technologies, you start by defining their goals and thoroughly documenting all the requirements. From there, you and your team can see what technologies will help you to reach the end goal.

Shaping a technology stack goes toe-to-toe with designing software architecture, so the engineers on your team should start by drafting a system design that will meet everyone's requirements. Software architecture is a wide and deeply technical topic, so we won't discuss it in depth in this chapter. Instead, we will present an overview of the necessary steps when it comes to choosing the best technologies for reaching specific goals.

Let's get started:

1. Collect the requirements and define the goals clearly.

2. Choose a set of architecture views for your project. An architecture view contains visual and textual descriptions of some of the aspects of the system. The most prominent examples of architecture views are as follows:

- Infrastructure view: Represents the physical parts of the system. Servers, storage arrays, and networks are documented inside this view.
- Component view: Represents the logical components of the software system that you are going to build. The component view should define the isolated parts of the system and the interfaces that they communicate through.
- Deployment view: Matches the logical representation of the component view with the physical reality of the infrastructure view. The deployment view should describe how the system components will be delivered to the corresponding hardware.
- Other views: Different methodologies for designing software architecture define many useful views. For example, the ArchiMate 2.1 specification defines 18 architecture views that can be useful for different stakeholders. For the sake of brevity, we will cover only the main views that can affect the technology stack and omit the others.

3. Define a list of necessary functions for technologies that you will use for the development and operations of your system based on the requirements and software design that has been produced by your team. Don't forget to include cross-cutting technologies that will optimize the experimentation, development, delivery, and operations of your system. Cross-cutting technologies may not help you find any specific functional requirements but will be beneficial to the project

in general. Experimental tracking frameworks, data version control systems, and Continuous Integration/Continuous Deployment (CI/CD) servers are all examples of cross-cutting technologies.

Example: let's build a technology stack for a customer churn prediction system. Our customer has defined the following list of requirements for the churn prediction module. We'll omit the technical details so that we can focus on the overall process:

1. Process the data from the marketing database. This contains customer information. The size of the dataset is under **5 GB** of raw data.

2. Train the churn prediction model on a weekly basis, every Monday. We should consider customers that have made no purchases for a month as churned.

3. Notify the team about the potential customer churn by using remote API calls. The service should be available on business days.

Decomposition classifies every requirement into two categories:

functional requirement (**FR**) and **nonfunctional requirement** (**NFR**).

FRs includes all the requirements that are related to the functions of the

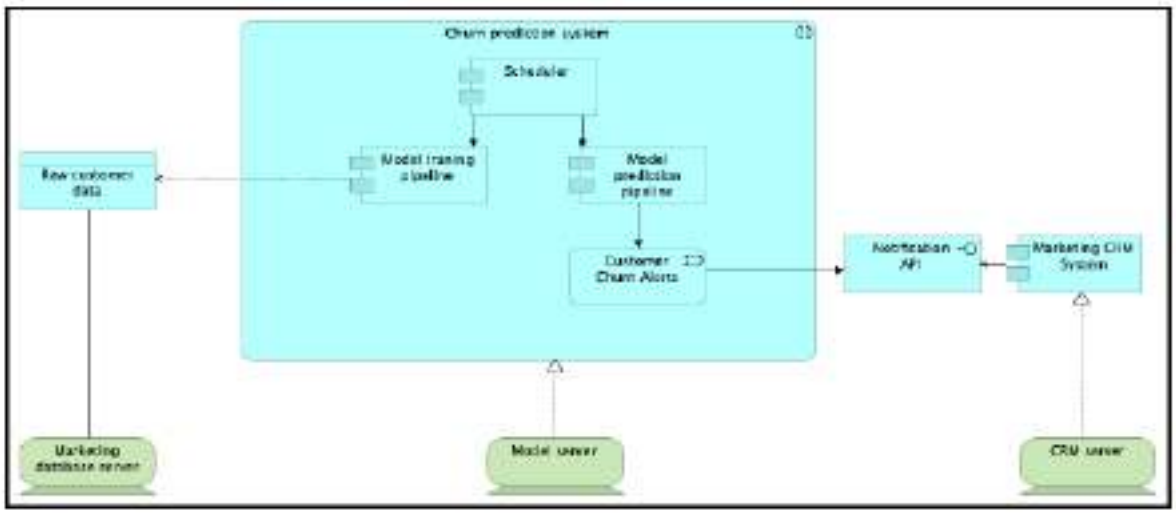
system that affect core use cases and end users.

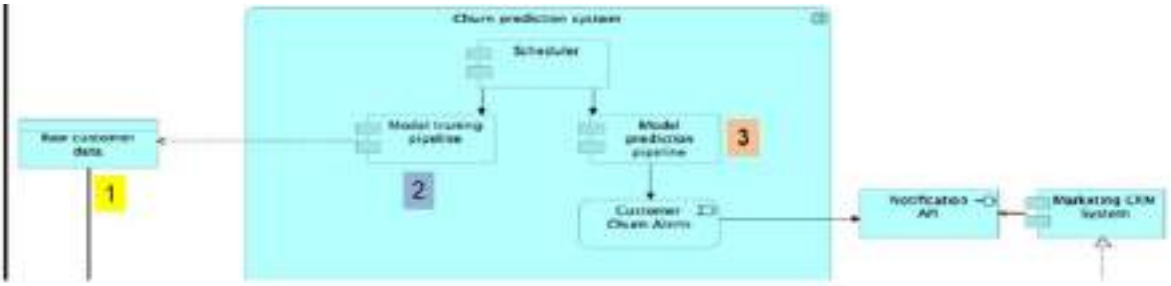
NFRs includes all the requirements that apply to the system in general and define a set of constraints in which the system will work. Service-level agreements and availability requirements are good examples of NFRs.

The team has decomposed customer requirements into the following list:

1. FR:
   - FR1: The system must be integrated with the customer's marketing database.
   - FR2: The system must provide a customer churn prediction model. We can consider customers that have made no purchases for a month as churned.
   - FR3: The system must call a remote API to notify the marketing department about potential customer churn.
   - FR4: The model should be executed every Monday.
2. NFR:
   - NFR1: The system should handle processing 5 GB of data every week.
   - NFR2: The API should be available on business days.

Based on this requirements list, the team has come up with the following system design, which has been drawn in ArchiMate 2.1 notation using the Archi software (`https://www.archimatetool.com/`):



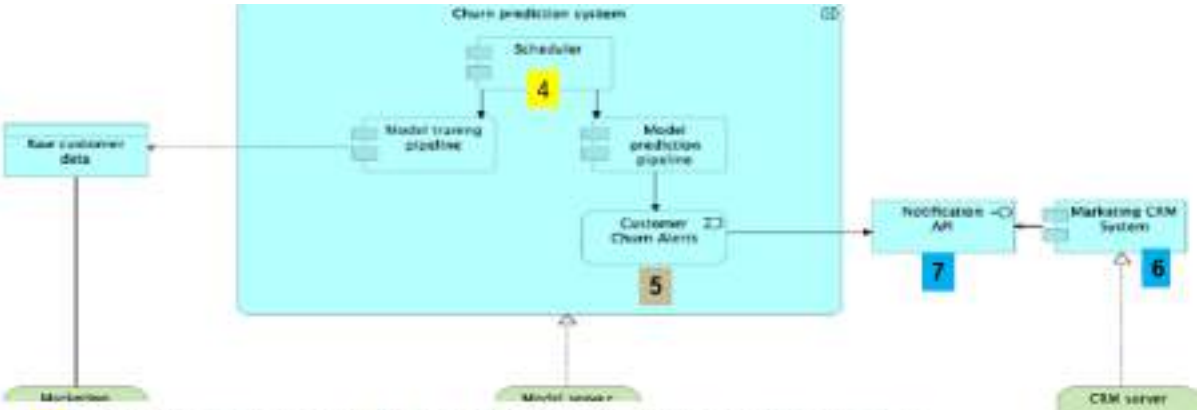


The software level describes the relationships between different components and services:

1 • **Raw customer data:** Represents the raw data that's used by the churn prediction system.

2 • **Model training pipeline:** Represents a set of data processing and model training steps, grouped as a software component.

3 • **Model prediction pipeline:** Represents the component that's responsible for taking trained models to make churn predictions and generate customer churn alerts.

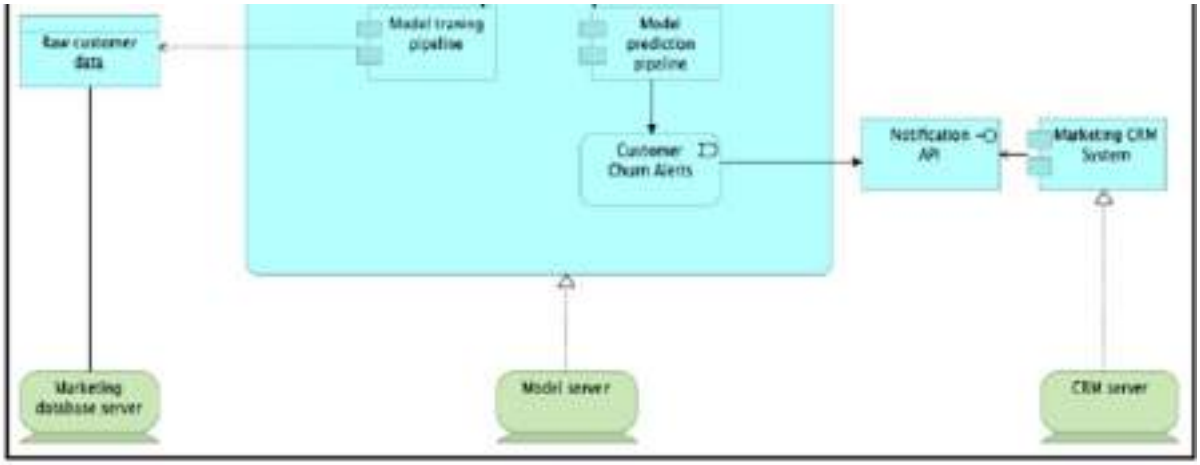


4 • Scheduler: Orchestrates the work of other components by running pipelines based on a schedule.

5 • Customer churn alerts: Notification events about potential customer churn.

6 • Marketing CRM system: The customer's CRM system, which has already been deployed and used in production.

7 • Notification API: A service that allows us to create notifications about customers inside the CRM system.



The infrastructure level, which describes the physical allocation of software to specific hardware resources, has three components:

- **Marketing database server**
- **Model server**
- **CRM server**

The preceding software architecture diagram omits many technical details for the sake of simplicity since our goal is to demonstrate how to make technology choices. The core idea of the following text is not to deepen your knowledge of the vast area of software architecture, but to give you a general feel for the process where requirements flow into sound technology choices. Understanding this process can help you guide your team of experts toward an efficient technology stack choice. The project did not seem very complicated to the team, and the task was fairly standard for them. The team has decided to use a general set of cross-cutting technologies that they use for every project as a corporate standard:

- Python as a programming language
- Git for source version control
- Data version control (DVC) for data version control
- GitLab CI/CD as a CI/CD server
- Jupyter Notebook for data analysis and visualization

To implement FR, they have decided to use the following technologies:

- FR1: The SQLAlchemy library for database access.
- FR2: scikit-learn as a machine learning library.
- FR3: Requests for API calls. The team has also decided to use a separate database to store prediction results and model execution logs.
- FR4: The team has decided to use cron (a popular Unix scheduler that can run commands based on a predefined scheduling table) as the main scheduling solution.

To implement NFR, they have decided to do the following:

- NFR1: Perform load tests and determine the minimum necessary server configuration for model training. From the team's previous experience, a virtual server with 8 CPU, 1 GB RAM, and 15 GB HDD should be sufficient, so they set this configuration as a baseline for tests.
- NFR2: The team has decided to request more detailed information on API usage and ask how many requests they can expect per day. It turns out that the API will be executed up to 10 times a day, so a single API server should suffice for this availability requirement.

Choosing between core- and project specific technologies

Technology choices should help with project requirement realization, but it is also crucial to take your team's expertise and capabilities into account, as well as the constraints. For example, if your team consists entirely of **Python** developers, choosing **Julia** as a primary programming language may be a bad idea, even if the team sees it as a better fit for the project:

- All of the team members will spend time learning a new language, practically destroying all productivity gains from using the technology.

The team's conclusions may be over-optimistic because of their lack of experience with the new technology.

Those two risks abate if your team pursues a growth mindset and gains new knowledge continuously, but they never vanish completely.

The core expertise in your team puts limits on what technologies you can use in projects. If you want more options, it is important to develop a team technology stack separately. Continuous internal research processes should keep your core technology step up to date.

Project-specific technology stacks can be formed by adapting the core technology stack for your project's requirements.

We can view the internal technology research process as a separate long-running project, which is best managed by using Kanban:

1. Someone on the team spots a potentially useful technology.
2. The experienced team members do a quick review of this technology. If technology looks promising, they put it into the internal research backlog.
3. Once the team manager decides that some time can be invested in internal research processes, they start a backlog grooming session. There, the team prioritizes tasks in the backlog and enriches their definitions according to the **specific measurable achievable relevant time-bound** (**SMART**) criteria.

4. The assignee takes a single task from the backlog and tries to finish it as quickly as possible. If they encounter a blocking problem, it should be instantly reported and solved, preferably with the help of other team members.

5. Once the research has been done, the assignee reports the results in the form of a document or a talk so that the team can decide on whether they will include the technology in the core technology stack.

6. If the decision is positive, it is crucial to create an educational program for wider internal technology adoption. The main result of this task is to prepare a workshop, guide, instruction booklet, or some other educational material that new and existing team members can use to familiarize themselves.

The core technology stack should not include overly specific technologies that are applicable to only a small set of projects. The sole purpose of this content is to help your team form a technological basis that solves the majority of requirements that arise in your projects. The more widely focused your team is, the more general the core technology stack should be. If the entire team is building a specific product, then project-specific and core technology stacks start to merge into a single entity.

The process of adapting the core technology stack into a new project is as follows:

1. Determine a set of project requirements.

2. See what requirements are satisfied by the core technology stack.

3. If some requirements are violated by the core technology stack, search for alternatives.

4. If some requirements are not met by the core technology stack, search for additions that can be integrated into the core technology stack.

5. At the end of the project, evaluate the new technologies that were added to the project and decide whether they are a good fit for the core technology stack.

Using these processes, you can turn often chaotic and desire-driven technology choices into a consistent and logical series of steps that lead to meaningful decisions. However, even the most deliberate requirement decompositions may leave your team wondering which technology to choose because there are many intersections and alternatives between different frameworks, libraries, and platforms.

Comparing Tools and Products

*Should we choose **R** or **Python**? What's better, **TensorFlow** or **PyTorch?***

A list of endless quarrels about which is the best **X** for doing **Y** can be found all over the internet. Those discussions are ceaseless simply because there is no silver bullet in the technology world. Every team of professionals has their specific use cases, which makes a certain technology choice work for them. There is no technology that will equally satisfy everyone.

X versus Y disputes often happen inside project teams, which is the most unproductive activity engineers can spend their time on. If you try to transition from X versus Y debates to searching for technologies that fit your specific needs (which are clearly stated, classified, and documented), you will get far more useful results in less time. Choosing the most modern or fashionable technologies is the closest analogy for playing Russian roulette for data scientists and software engineers. Let's explore how we can make thoughtful decisions about technology stacks.

To make meaningful technology choices, you need to make the process more systematic. First, you need to derive a set of comparison criteria that will allow you to benchmark different technologies and provide a template for research activities. These criteria should test different dimensions or groups of requirements that make the technology useful in your specific case.

Case study – forecasting demand for a logistics company

Let's imagine that we need to choose a time series forecasting framework for our project. Our team works primarily in Python and has to provide a tool that will provide forecasts for time series data. The main goal of this system is to forecast demand on a set of given products that are shipped by the company. The team has discovered that there are many different forecasting frameworks.

To choose between these products, the team has created the following comparison criteria:

ID	Requirement definition	Substantiation	Score	Priority
		Ease of development		
D1	Compatible with Python	Python is the major programming language for the team.	3	Mandatory
D2	Compatible with scikit-learn interfaces	The team has good expertise in the `scikit-learn` library. It would be beneficial if the time-series forecasting library were compatible with `scikit-learn`.	2	Important
D3	Has good documentation		1	Supplementary
D4	Forecasting can be done in under 10 lines of code		1	Supplementary
D5	Can handle timeseries as a native data type	The framework should work with raw time series data so that the team doesn't spend additional time on dataset preparation and feature engineering.	2	Important
		Forecasting algorithm requirements		
F1	Does not require manual hyperparameter tuning	Since the library will be used with a large number of time series, manual model tuning is not practical. The tool should support automated hyperparameter tuning or be robust when it comes to which hyperparameter is chosen, thus providing good forecasts with the default settings.	3	Mandatory
F2	Provides several forecasting methods	The availability of several forecasting methods will allow us to evaluate several models and choose the one that fits the best for each specific time series.	1	Supplementary
F3	Works with seasonal time series	All the time series in the customer data have seasonal patterns.	3	Mandatory

F4	Provides confidence bounds, along with forecasts	Confidence intervals can be used to provide uncertainty bounds for each forecast, which the customer considers a useful feature.	2	Important
Performance and data requirements				
P1	Forecasts for time series with 100 data points can be done in under 15 seconds	A large number of time series limits the total amount of time we can spend on a single time series.	2	Important
P2	Can handle time series with a variable length and empty data	Data quality is not ideal and some gaps are present in the data. Some items have more historical data than others.	2	Important

The preceding comparison table consists of the following columns:

- **ID**: This can be used as a short identifier in the technology comparison table (provided next).
- **Requirement definition**: This should describe the capability of interest.
- **Substitution**: This should provide the motivation behind the requirement.
- **Score**: This shows the relative importance of the requirement and will be used to summarize each requirement category.
- **Priority**: This indicates the necessity of each requirement and will provide additional scores for each technology.

Next, the team has prepared a list of frameworks to compare, as follows:

- **Pmdarima** (`https://www.alkaline-ml.com/pmdarima/`)
- **statsmodels.tsa** (`https://www.statsmodels.org/stable/tsa.html`)
- **Prophet** (`https://github.com/facebook/prophet`)
- **LightGBM** (`https://lightgbm.readthedocs.io/en/latest/`) **and tsfresh** (`https://tsfresh.readthedocs.io/en/latest/`)

The team has also come up with the following comparison table:

Framework	D1 (M)	D2 (I)	D3 (S)	D4 (S)	D5 (I)	F1 (M)	F2 (S)	F3 (M)	F4 (I)	P1 (I)	P2 (I)
pmdarima	3	2	0	2	3	3	0	3	2	0	0
statsmodels.tsa	3	0	2	0	3	0	1	3	2	2	0
Prophet	3	0	2	2	3	3	0	3	2	0	2
LightGBM and tsfresh	3	2	2	0	0	0	0	3	0	2	0

This table can be further summarized to give you the following final results:

Framework	Ease of development score	Forecasting algorithm requirements score	Performance and data requirements score	Mandatory requirements satisfied	Important requirements satisfied	Supplementary requirements satisfied
pmdarima	10/12	8/9	0/4	3/3	3/5	1/3
statsmodels.tsa	8/12	6/9	2/4	2/3	3/5	2/3
Prophet	10/12	8/9	2/4	3/3	3/5	2/3
LightGBM and tsfresh	7/12	3/9	2/4	2/3	2/5	1/3

If you wish, you can use weighted averages to summarize each framework's score into a single number. However, remember that simplifying complex decisions into one number can lead to errors if this is done incorrectly.

These research results show that Prophet comes up as a better choice according to the initial requirements. However, the results do not mean that Prophet is the best choice for every application. Technology choices should be biased and opinionated since no technology can be the best fit for every project. For example, the ranking could be entirely different if the team took the desired average metric value into consideration. In this setting, other frameworks could have won because they provided more accurate models.

End of Chapter 12

End of session 6