|  |
| --- |
| IU |
| Managing Data Projects |
| Course Code  Rafal Wlodarski  If you do not wish to be credited, please indicate it below:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

# Learning Objectives

This coursebook aims to equip you with a variety of skills (both soft and technical), technologies, and tools to efficiently manage data-intensive projects. First, you will learn agile project management principles and the Scrum framework, which provides an overarching structure for a collaborative, iterative, and incremental way of working.

Afterwards, modern approaches to automation of the development lifecycle, as well as their supporting technologies, will be covered so that you understand their purpose and know when to apply them. Additionally, almost every chapter provides you with opportunities to get hands-on experiences with DevOps practices, like automated testing, continuous integration, continuous delivery, infrastructure as code and containerization.

By the end of the coursebook you will have both a high-level, organizational view of data-intensive projects and a deep understanding of the underpinning technologies and tools. Only the combination of both will make you a valuable team member of such endeavors regardless of your role – whether you are the person in charge or contribute as a developer or data scientist.

# Unit 1 – Agile Project Management for Data-intensive Projects

**Study Goals**

On completion of this unit, you will be able to …

… implement a project management process based on CRISP-DM.

… describe agile values and contrast them with traditional project management.

… apply Scrum to efficiently manage data-intensive projects.

… recognize the challenges of traditional and agile approaches in the context of data-based solutions.

# 1. Agile Project Management for data-intensive projects

## Introduction

As has been said, “*Necessity is the mother of invention,*” and *agile* approaches emerged as a response to the need for modernization of software project management. Early computer-related projects relied heavily on hardware and were characterized by relatively clear requirements – hence **plan-driven approaches** to management that sought predictability, stability, and reliability were widely employed. Ever since, the information technology industry’s focus shifted to software and then to data, which called for new methods of project management.

**Traditional software project management**

(classical/*plan-driven*/heavy methods), Dated family of approaches (e.g., the Waterfall model) that consisted of sequential phases, such as analysis, design, implementation, test, and deployment, with well-defined resulting artifacts. They relied on the assumption that requirements can be clarified upfront and expected project stages to be fully finished before moving on to the next one (without the possibility to revert back).

New theories, methods, and approaches from the agile family were gradually born in the 1990s. Crystal was the first formally formed agile method in 1991, followed by Scrum in 1993 and many others years later. While there might not be a one standard definition of agile project management, it can be clearly distinguished from traditional methods, which are characterized by lengthy and sequential project phases.

The Cambridge Dictionary defines agile as “able to move quickly and easily,” and indeed methods from this family were conceived to rapidly adapt to changes – be it in market, technology, or environment – and focus on early delivery of business value.

To congregate all the features and characteristics that agile methods have in common, in 2001 a group of project experts gathered together and created the **Agile Manifesto** (Beck et al., 2001), a statement of values for successful project development. These core principles are as follows:

* “individuals and interactions over processes and tools”
* “running software over comprehensive documentation”

**Data science**

An interdisciplinary field that aims to create value through the analysis of data. It encompasses (Saltz, 2021) a myriad of related terms such as big data, business intelligence, business analysis, machine learning, and finally, data science. In the context of this book, it is used to denote a discipline where projects revolve around data.

* “customer collaboration over contract negotiation”
* “responding to change over following a plan”.

However, to fully understand the shift from traditional project management to the “new way of thinking” that was introduced by agile, one needs to be familiar with the classical life cycle of data-intensive projects. In such projects, data are the most important challenge – be it the sheer quantity that need to be processed, the complexity of the data themselves, or building intelligence based on them – and this has implications on the way of working. The following section provides an overview of a process model in **data science** as embodied by CRISP-DM – the most commonly used method for project management in data-intensive endeavors (Piatetsky, 2014; Saltz, 2022). Despite its dominating presence on the market, it fails in some facets of project management. Such shortcomings are addressed by the agile approaches, as you will learn later on.

## 1.1 Data Science Life Cycle

CRISP-DM, which stands for Cross-Industry Standard Process for Data Mining, was introduced in the 1990s and has become an industry-proven way to guide projects in the data science field (Piatetsky, 2014; Saltz, 2022). As a methodology, it provides descriptions of the typical phases of a project, the underlying tasks, and their relationships. As a process model, it is an outline of the data science life cycle as depicted in the diagram below (arrows indicate the most important and frequent dependencies between phases).

Data Science project life cycle in CRISP-DM

Modeling

Evaluation

Deployment

Data Preparation

Data Understanding

Business Understanding

DATA

Source: Rafal Wlodarski (2023), based on CRISP-DM Help Overview, IBM Corporation, [https://www.ibm.com/docs/en/spss-modeler/saas?topic=dm-crisp-help-overview]

The life cycle model consists of six phases: business understanding, data understanding, data preparation, modeling, evaluation, and deployment.

### Business Understanding

This initial phase aims to frame the project by determining business drivers and functional expectations. This stage is of paramount importance as the absence of specific business objectives and clear requirements can result in an improper scope of the project and jeopardize its chances of success from the very beginning. The usual tasks that underpin the business understanding phase are as follows:

1. **Determine business objectives**: Get a proper understanding of what your client is trying to achieve, why the project is needed and what expected benefits the project will deliver upon its completion.  The *business success criteria* are inseparable as they allow one to judge its achievements and objectively evaluate the project’s outcomes.

2. **Assess situation**: This is an umbrella term for many operational activities that are necessary to kick off the project. This includes determining resource availability and making sure you have the right team for the job, verifying technical, security, and legal requirements for the project, and listing its constraints and assumptions. You also need to identify potential risks throughout the entire project and methods to mitigate them.

3. **Determine technical project goals**: With the business objectives clarified, it is necessary to translate them into the technical domain. Determine the type of the problem (prediction, classification, etc.) as well as techniques applied to solve it, and provide actual numbers for desired outcomes.

4. **Produce project plan**: A timeline of your undertaking is a reference point to assess the current situation, serves as a guide towards the project completion, and should indicate any milestones and decision points along the way. Account for and mark any activities that may involve multiple iterations, such as modeling. Select technologies and tools that will help you deliver the project.

To achieve the above tasks, a series of meetings (both in person and online), interviews, documentation reading, and observations of the work environment may be required to ask the right questions about your project context. The list is not exhaustive and can be extended with any field-specific learning activities.

### Data Understanding

Building on the team’s business understanding of the problem, the objective of this phase is to identify what data will be needed as well as where and how to acquire them to address stakeholders’ expectations. The team strives to extract the best value from existing data sets and identify, collect, and analyze new sets of data for the project. Should the relevance of some data be unclear, momentarily reverting to business understanding activities is appropriate.

The standard tasks that deal with data understanding are as follows:

1. **Collect initial data**: Gather the existing and acquire any supplemental data needed from external sources or by own means (surveys, interviews, tracking additional data).
2. **Describe data**: Determine the quantity of data, and their types and properties. Examine the data and document their relevant attributes, such as data format, number of records, or field identities.
3. **Explore data**: Learn more about your data by querying them, creating different representations – tables, charts – to work out relationships between data, reveal new characteristics, and identify attributes that seem promising for further analysis.

**Metadata**

Data describing other data. Examples include origin, date, or time.

1. **Verify data quality**: Perform thorough quality assessment to identify any inconsistencies and avoid ensuing work on incomplete or skewed data. Common types of issues that should be detected at this stage are as follows (IBM, 2011): missing data, data errors (such as spelling mistakes), coding inconsistencies (nonstandard units of measurement or value inconsistencies), or incorrect **metadata**.

### Data Preparation

This phase of a data-intensive endeavor is crucial and usually represents somewhere between 50 and 70% of the whole project in terms of duration. Most commonly, the following four tasks are undertaken:

1. **Data selection**: Using the initial data collection (outcome of the first task of the previous phase), an informed decision is taken in terms of choosing specific data sets that will allow meeting technical project goals. The selection concerns items (rows of data) and their characteristics (columns). It is essential to document the rationale and reasoning that drove major decisions taken during data selection.
2. **Data cleaning**: This is a process that involves fixing or removing data items that have quality issues. There are various problems that can occur and methods to deal with them (see a table below for an overview). The goal of the data cleaning step is to ensure that the concept of “garbage in, garbage out” doesn’t become reality on your project. Additionally, to save processing time during the modelling phase, format your data to comply with the target models of your project (particular data formats, such as string/numeric or order).
3. **Deriving new data**: Whenever new attributes can be useful, add columns that allow you to capture such information. Ask yourself, based on your knowledge, if there are any facts that can be derived using existing fields (e.g., BMI index from height and weight fields)? Similar action can be taken whenever generating additional data brings value.
4. **Integrating data:** Multiple sources of data can be merged to generate a new data set when they share a unique identifier and answer the same business questions (e.g., mortgage history and demographics information; shared key: social security number). Combining two or more data sets based on their attributes is also possible by appending them using a similar field (e.g., product name).

The deliverables of this phase include the following: data selection decisions record (with rationale), data cleaning report, documentation of merged/reformatted data sets.

|  |  |
| --- | --- |
| Data Quality Issues and their Corresponding Data Cleaning Methods | |
| Data problem | Possible solution |
| Missing data | Exclude rows or columns whenever appropriate; alternatively provide an estimated value. |
| Data errors | Use logic to locate issues and replace errors; alternatively exclude columns whenever appropriate. |
| Coding inconsistencies | Choose only one coding scheme and apply to any values that don’t respect it. |
| Missing or incorrect metadata | Manually inspect fields that are of concern and uncover correct meaning. |

Source: Rafal Wlodarski (2023), based on IBM (2011)

### Modeling

The modeling phase is often seen as a core of data-intensive endeavors as this is when all prior efforts start to produce tangible results that address the business problem posed. Most commonly, this phase is carried out iteratively, meaning that models are built, tested, and evaluated in steps, until a satisfactory solution is found. This implies running several models with their default parameters, analyzing the results, and fine-tuning parameters or choosing other models.

The modeling phase consists of four major tasks:

1. **Selecting modeling techniques:** While keeping your business goals in mind, a decision needs to be made on which model (e.g., regression, neural network) is the most appropriate for your project.
2. **Generating a test design:** Testing a chosen model allows you to determine how “good” it is; defining the criteria that constitute that characteristic is part of the job. Depending on whether a model is supervised or not, various attributes may be relevant: the error rate, required processing time, or ease of deployment. Therefore, generating a test design involves defining success criteria for a model and choosing data that will help evaluate them.
3. **Building a model**: technically this task might only involve running a model with a set of parameters using a software/tool or executing a few lines of code. The non-obvious part is proper documentation of the parameter settings and the model results, including any performance and/or data-related issues that might have occurred.
4. **Assessing a model**: in the modelling phase, multiple models are used and evaluated so that the most accurate or effective one for the solution can be chosen. A methodical assessment based on domain knowledge, success criteria, and the test design are the right way to make such a call. Given the importance of this decision, you might want to follow a detailed task list such as the one depicted on the diagram below.

Model Assessment Task List

Source: Rafal Wlodarski (2023)

### Evaluation

By now, the lion’s share of the work is behind the team, and management needs to ensure that the business needs are met with the models produced. To do so, three major tasks are carried out:

1. **Evaluating the results:** Key decision makers formally assess whether the project results meet the business success criteria. The list of proposed models needs to be evaluated and, if time allows, tested on a practical use case. Every model should explicitly state whether it addresses the business goals established in the business-understanding phase. Before such a meeting takes place, an internal check list may come in handy (see below).
2. **Reviewing the process:** Effective project management methods include activities that allow one to reflect on the process employed. Such activities are at the heart of Scrum and will be covered in detail later on.
3. **Determining the next steps:** Based on the output of the previous two tasks, a decision needs to be made whether to continue to the final project phase or whether refinement/replacement of certain models is necessary.

### Deployment

Deployment implies making the results of your data-intensive project available and usable by customer stakeholders. Depending on the context, it might mean formal integration, such as implementation of a real-time predictive model, or simply sharing a report with findings based on a model’s results. Regardless of the size of the undertaking, the following four tasks need to be executed:

1. **Planning deployment:** Create a document that provides step-by-step instructions for deployment and integration with existing systems/infrastructure.
2. **Planning monitoring and maintenance**: Certain models need to be evaluated periodically to make continuous improvements. In order to do so, determine how the accuracy of a model can be measured, which factors should be tracked to enable this, and at what frequency.
3. **Producing a final report**: The project team documents a summary of the project to present the results and the means to you took to arrive there to various (both technical and business) stakeholders. A typical structure of a final report can be found in IBM (2011, ch. 7 – Producing a Final Report).
4. **Review the project**: The final step of the process allows you to collect lessons learned to improve in the future.

Check List for Business Goals Evaluation

Source: Rafal Wlodarski (2023), based on IBM (2011)

### Challenges of a Traditional Data Science Life Cycle

While the CRISP-DM process is easy to understand, intuitive, and provides a relatively detailed set of steps to follow in a data-driven project, it has multiple shortcomings.

First, it doesn’t address the communication aspects of a project, either within a team or with the client. It is not prescriptive on how the team should coordinate (a crucial aspect of a large-scale undertaking) or collaborate with the stakeholders (consequently minimizing their input throughout the project). The focus is mostly on technical solutions and producing documentation, which does not translate directly to business value.

Finally, the importance of testing is overlooked, which results in late discovery of bugs and delayed client feedback. While CRISP-DM covers a lot of universal knowledge for data-driven projects, it has not been updated since 1999 and is criticized for not meeting the considerations of big data and modern data science projects. Therefore, agile methods have gained more and more traction in the sector as they address many of the weaknesses outlined.

### Self-Check Questions

1. Which phase of a Data Science project life cycle in CRISP-DM usually takes the longest to complete? Does it mean that it requires the most planning from the team?

Data Preparation; no, all phases are equally important and are expected to be planned upfront with care.

## 1.2 Scrum Theory and Values

In agile methods, the importance of the relationship, spirit of cooperation, and belonging of the project team is paramount. The human role in the process of development is far more significant than tools and institutionalized processes. What this means in practice is that people can respond more quickly and transfer ideas more efficiently via direct interactions rather than through reading and writing documentation.

Secondly, the crucial point of agile methodologies is to continuously provide a tested and working solution or modules thereof. Releases are carried out frequently, varying from once a month to even once a day. Agile methods embrace the idea of simplicity, where no more code is produced than necessary and no documents attempting to predict the future are created, as they inevitably become out-of-date.

Third, participation of the client in the project and close cooperation with the project team is critical and more important than strict terms of a contract. A client whose needs are well understood and addressed is a satisfied client, and at the end the expected business value is delivered along with a fulfilled contract.

Fourth comes the assumption of project transparency. This implies that both the project team and customer representatives are well-informed and have the necessary knowledge and skills to make changes to the predicted plan as the development of the project progresses. Requirements changes are inevitable, and the real difficulty is to react appropriately and handle them well.

The authors of the Agile Manifesto complemented it with 12 principles that provide additional detail on agile development. From there on, those basic concepts were further enriched by supporting agile practices (Jalali et al., 2010), which were devised to help accomplish agile principles in a given project management method.

Core Values of Agile Methods

Source: Radal Wlodarski (2023)

1. **Our highest priority is to satisfy the customer through early and continuous delivery of valuable product**: One should deliver every couple of weeks consecutive, more functional versions of the product to obtain feedback from the client and make sure what was already developed complies with their needs.
2. **Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage**: Flexible structure of the developed product minimizes influence of modifications on the shape of the entire project.
3. **Deliver working artifacts frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale**: Such an approach empowers regular verification of the product, finding issues as well as getting early feedback from the customer.
4. **Business people and the project team must work together daily**: Close cooperation between the customer and project team is imperative to fulfill all of the requirements and distinguishes the agile family from traditional methodologies.
5. **Build projects around motivated individuals, give them the environment and support they need, and trust them to get the job done**: As recognized multiple times, people are the most important part of the project; therefore, necessary working conditions and assistance need to be provided to them.
6. **The most efficient and effective method of conveying information to and within a project team is a face-to-face conversation**: This is irreplaceable – the best means of communication is a direct discussion.
7. **Working artifacts are the primary measure of progress**: Size of the documentation or number of lines of code are far from being representative indicators of progress, and nothing illustrates success better than functionality that is in line with the customer’s needs.
8. **Agile processes promote sustainable development; the sponsors, developers, and users should be able to maintain a constant pace indefinitely**: A stable and tailored pace is more effective than demanding maximal efficiency from the employees at all times.
9. **Continuous attention to technical excellence and good design enhances agility**: Applying proven solutions and sticking to good practices from the very first iterations of the project enables the avoidance of many mistakes and empowers agility.
10. **Simplicity**: The art of maximizing the amount of work *not* done is essential; developing extra functionalities, which sometimes the customer is not even aware of, is usually a waste of time, and it is advised to focus on what is necessary.
11. **The best architectures, requirements, and designs emerge from self- organizing teams**: The responsibility for tasks should lie on the entire team and not on particular members; it is highly important for every member to be informed and knowledgeable about others’ work.
12. **At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly**: Working environments are constantly changing, which implies that the team itself needs to be flexible, able to respond and adapt to these changes.

### Scrum method

Scrum is nowadays the most commonly used agile approach (KPMG, 2017) for software development, but it dates back as far as 1986, when it was defined by H. Takeuchi and I. Nonaka (see Takeuchi, 1986 for more details) as “a flexible, holistic product development strategy where a development team works as a unit to reach a common goal.” The authors did not employ the term Scrum, but instead referred to it as a “holistic” or “rugby” approach, because the whole process is performed by one cross-functional team during multiple overlapping phases.

Their work was adapted to the software development domain by Ken Schwaber and Jeff Sutherland in 1995 (see Schwaber, 1995 for more detail) and later refined and documented in the Scrum guide (Sutherland, 2020), which is the present-day reference on Scrum.

The software development process as guided by the Scrum methodology is depicted below and is based on the following concepts:

1. Three roles:
   1. Product Owner
   2. Scrum Master
   3. Development team.
2. Five recurring events:
   1. Sprint
   2. Sprint planning meeting
   3. Daily scrum (stand-up meeting)
   4. Sprint review
   5. Sprint retrospective.
3. Three development artifacts:
   1. Product backlog,
   2. Sprint backlog,
   3. Increment.

Structuring concepts of the Scrum method

Graphical user interface, diagram

Description automatically generated

Source: Rafal Wlodarski (2023), based on Kneuper (2018).

### Self-Check Questions

1. What does it mean to work iteratively?

*Work in time-boxed intervals*

1. Please fill in the gaps in the sentence that represents one of Agile key principles: \_\_\_ and interactions over \_\_\_ and tools (*individuals*, *processes*)

## 1.3 Scrum Teams in Data-Intensive Projects

Agile software development methods emphasize teamwork and recognize its human aspect, and Scrum is no different in its team-focused definition of primary roles on a project: Product Owner, Scrum Master, development team.

A Scrum Master is a guardian of the methodology and ensures that the team understands and fully follows Scrum principles to achieve business value rather than administering the group. He/she educates and guides both the Product Owner and the development team in the skillful use of Scrum. The Scrum Master is the enforcer of the rules of Scrum, its practices, and the corresponding process.

A Product Owner is the empowered, central point of product leadership who determines which features and functionalities should be delivered, in which order and where to focus observation and analysis efforts. These elements are translated into a prioritized list, the so-called product backlog, which serves as input for the project planning. A Product Owner represents the “business side” of the project as he/she strives to maximize the return on investment (ROI) and satisfy all the stakeholders.

The development team is responsible for bringing the product to life, which comprises all relevant tasks, such as designing, coding, testing, and documentation. The size of the entire Scrum team should not exceed ten people with three up to nine development team members, one Product Owner, and one Scrum Master constituting the Scrum team. The composition of a Scrum team was designed to optimize the flexibility, creativity, and productivity of its members and by design should not be altered. Project scaling is done through having teams of teams rather than increasing the number of resources in one team.

Typical Roles in a Data-Intensive Project

Source: Rafal Wlodarski (2023) based on R. Jurney based on Agile Data Science 2.0 (2017)

Data science is a broad discipline, which spans activities like analysis, design, development, business, research, and operations. While traditionally, every person would perform only one role, a Scrum team is said to be **cross-functional**. This implies that all team project members have the skills needed to analyze business requirements, create artifacts (e.g., models, mockups) and carry out experiments (design, build, test, and deploy the desired product).

What is more, a Scrum team is a self-organizing entity that has autonomy to determine how to execute its work as long as the objectives that the team collectively committed to are met. This means there is no project leader or any other leadership role as defined in classical project management. However, in practice, many organizations do not follow the rules of “cross-functional” and “self-organizing” teams and distinguish various roles that commonly support data-intensive projects as depicted in the figure above.

### Self-Check Questions

1. Which role(s) in Scrum are involved in planning activities and to what extent?

*A Product Owner influences the scope of a Sprint by prioritizing which functionalities should be delivered. The development team determines how many functionalities can be implemented and how best to plan the implementation process.*

## 1.4 Scrum Events in Data-Intensive Projects

### As one of the representatives of agile methods, Scrum is an iterative approach, with work being structured in entities called ***sprints***. They are commonly a couple of weeks long, but should never exceed a month. Moreover, they are of fixed duration, finishing at a specific date, even if the planned objectives have not been met.

**Iterative, incremental**

An iteration is simply a sequence of steps that are executed repeatedly in a given order. Increments successively add desired functionalities of the final product so they deal with content as opposed to time (for iterations).

### Getting started

Each sprint begins with a ***sprint planning meeting***, where members of the Scrum team select several items from a prioritized list of requirements and commit to implementing them in the upcoming sprint. This subset of the project’s functionalities is meant to materialize as a new increment of a working, potentially shippable product.

The meeting lasts roughly two hours for a two-week sprint and four hours for a four-week sprint. Presence is mandatory for the Product Owner and the project team; the Scrum Master is usually there as well, and other stakeholders may attend the meeting, though this is rare.

The main task of the Product Owner is to communicate their goal for the sprint and prioritize the items from the product backlog they want to have implemented. Then, the team and the Product Owner discuss the high-priority items, clarify the requirements, and decide which functionalities can be delivered during a particular sprint (long-term goals and tasks are not encompassed by this meeting).

Once the items are made clear and agreed upon, the team can do a detailed estimation of the tasks’ complexity and eligibility for the sprint. They start with the priority items and work their way down the list until they have used the total capacity available for the sprint.

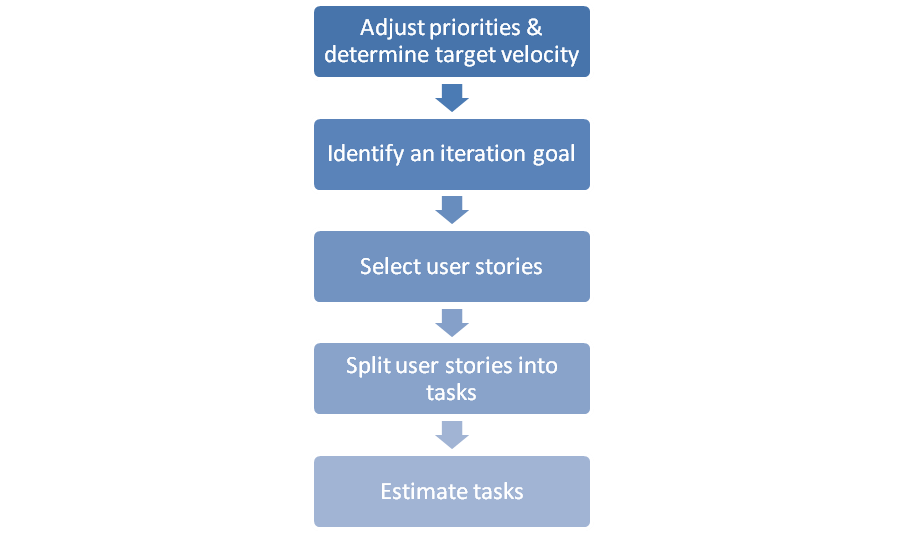
To be able to estimate the possible work to take on, each planning session is based on the team velocity, a straightforward concept of the sum of the estimates (in story points or any other unit used by a particular team) of delivered features per iteration. It accurately measures the workload that a team can perform during one sprint by indicating a number of points that all tasks’ estimates summed up cannot exceed.

**Story points**

are a unit of work used for estimation in Scrum. As opposed to man-days, they don’t translate to time but represent the complexity of a given task, relative to other tasks. Very often Fibonacci numbers (1, 2, 3, 5, 7, 13, 21..) are used as they don’t increase linearly.

Team members’ availabilities during a sprint, vacations, and other impediments need to be incorporated in the calculations as they decrease the potential work to be done.

The Main Steps of a Sprint Planning Meeting

  
Source: Rafal Wlodarski (2023)

A rule that says that goals mustn’t change during a sprint’s duration and must be postponed until the next sprint planning meeting is a foundation of the Scrum methodology. The Product Owner is not allowed to introduce new items for the team to work on halfway through the sprint once a commitment has been during the sprint planning meeting [14]. In case external interference significantly influences the workload planned, the Product Owner can terminate the current sprint, carry out a sprint planning meeting incorporating the new circumstances, and start working within the next sprint.

### Working with Scrum

Once the sprint has started and the team gets to work, it engages itself in one of the key Scrum practices – daily scrum. This is an every-day, mandatory meeting of the entire project team (the Product Owner can attend along with other interested parties). It is 10 to 15 minutes long and usually takes place at the same time and location each day. Often it is referred to as a daily standup meeting, because all participants are standing, which keeps the meeting to the point and within the planned timeframe. Each participant answers the following questions:

* What did you do yesterday?
* What will you do today?
* Are there any impediments in your way?

By answering these questions, each team member knows what other team members accomplished the day before and which tasks they plan to conclude before leaving the office. The team gains awareness and understanding of the project’s short-term progress and what quantity of work remains unfinished.

While everyone is asked about the obstacles they encountered, daily standup is by no means a problem solving or issue resolution meeting. Problems are raised, but they are not dealt with until after the meeting, so that the people not directly involved in the issue do not waste time or get bored.

It is the Scrum Master’s responsibility to take care of all the impediments and overcome them as soon as possible so that the team members can continue their work. While usually these involve non-technical issues that the Scrum Master can deal with, they can also delegate the problem to another team member or seek external help as long as the issue is resolved rapidly.

While the team progresses with work during a sprint, before any task can be considered fully completed and delivered as part of an increment, it needs to respect a “**Definition of Done**.” This represents criteria to be met and usually covers the aspects of coding standards, unit tests, integration, and documentation and is a determinative quality reference point to the team. Although, as in data-driven projects, it is common to discard some code (an experiment needs to prove useful to retain a given technical solution), the constraints on its quality may be more relaxed than in a software project.

When a sprint comes to an end, project stakeholders are invited to a ***sprint review meeting***, which takes the form of a demonstration, and provide feedback on the produced increment of the product. Release of a production-ready increment of the product at the end of each sprint is one of the fundamental concepts of Scrum.

This meeting has rather informal character and it is highly recommended for the team not to spend more than two hours preparing for it. Presence is mandatory for the Product Owner; all of the team members involved in the demonstrated functionalities and the Scrum Master, with customers, stakeholders, and anyone interested are welcome to join.

The closing activity for the cycle of development in Scrum is a **sprint retrospective**, which is usually carried out right after the sprint review. It is of high importance and should never be skipped as it enacts the “*inspect and adapt*” characteristic of Scrum. Scrum stresses the importance of taking a short step of development, reflecting on it, and inspecting the resulting product and the efficiency of current practices. In the ordinary course of events, it is the team members and Scrum Master that participate in the meeting as its main goal is to identify improvement opportunities from an organizational, project management point of view and to tweak the Scrum process. Every person present is expected to contribute with points listed on the diagram below. A discussion takes place afterward to identify what needs ameliorating and decide on the changes to introduce in the following sprint.

A Common Approach to Organize a Sprint Retrospective – a Whiteboard with Three Columns

Source: Rafal Wlodarski (2023)

### Self-Check Questions

1. What’s the difference between daily scrum and stand-up meetings?

*They’re the same*

1. Does Scrum include activities that foster continuous improvement? If so, which one(s)?

*Retrospective*

## 1.5 Scrum Artifacts in Data-Intensive Projects

The first stage of every project is *requirements gathering*, and Scrum is no different in this respect. However, as opposed to traditional methods of project management, only primary requirements are defined. The reasoning behind this is that requirements tend to change and evolve during the development process, hence there is no need to specify all functional aspects beforehand.

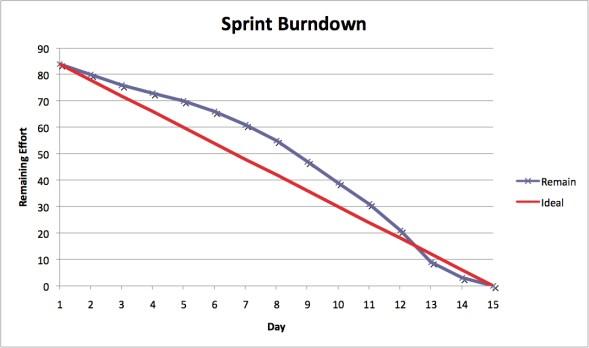
The most common way to express the features of a product is in terms of **user stories**, which are clear and concise characterizations of the desired functionality, told from the perspective of the user. An illustrative example might be the following: “As a marketer, I want to call customers with a high likelihood of accepting an offer.” A common practice is to write them on sticky notes or index cards and arrange them on the wall of the project room to ease planning and discussion. User stories do not represent technical tasks (these are defined by the team, during a sprint planning meeting) but focus on the value from customer’s point of view.

A refined and prioritized list of requirements is called the **product backlog**, and this is the heart of Scrum. Items listed in the product backlog, also referred to as features, describe customers’ needs captured by the Product Owner, and they describe functionality aspects that are potentially to be delivered. The primary version of the product backlog includes only the well-known and understood requirements in a prioritized manner, so that they can be implemented in the first sprint. It is allowed to evolve and grow over the lifetime of the product, as more and more is learned both about the product and its customers.

Whenever the team selects a subset of requirements from a product backlog for implementation, they are moved to a **sprint backlog**. This represents a selection of functionalities that the team commits to develop as part of a given sprint. Technical works, which may represent improvement goals (e.g., rework a module to increase security) or necessary actions before development (e.g., upgrade all workstations to Windows 10), knowledge acquisition tasks (e.g., research available technologies and choose the best one in terms of performance), or known bugs and defects (e.g., fix the encoding of characters) can also form part of a sprint backlog.

The artifact, once produced in the sprint planning meeting, needs to be updated on a regular basis – ideally each time a new information is available; however, not less frequently than daily. With updated information on the amount of time remaining to finish particular tasks, a sprint *burndown chart* can be generated. By summing up the remaining story points for the whole team and plotting them against a time axis, one can clearly see the remaining to complete the current sprint. While usually it is not a straight line that reaches zero effort remaining (x-axis crossing point) by the last day of a particular sprint, which would indicate a perfect workload plan and no impediments to the development, it motivates the team by showing how they are progressing toward the goal and how much work they have left. The figure below shows a burndown chart for a real-life Scrum development, where in the first days of the sprint, the predicted effort is exceeded, and the team seems to lag behind; however, by the end of the development, all of the tasks are finished and the “zero effort” point is crossed.

An Example of Sprint Burndown Chart



Source: Rafal Wlodarski (2023)

Apart from the burndown chart, there is an additional, straightforward tool used to keep track of the workload during a Sprint: the **scrum task board**. This is a physical or digital board, where all of the tasks in the form of post-it notes from current sprint are gathered and grouped in columns, representing the different statuses of the tasks (to do, in progress, testing, done).

Each sprint needs to produce at least one product **increment**, which encompasses multiple working functionalities and represents concrete customer value. Every increment is additive to all prior increments, which implies that thorough testing is required to ensue all increments work together.

### Self-Check Questions

1. What are the type of backlogs that exist in Scrum and which one contains more requirements?

*The product backlog and Sprint backlog; the former always has more requirements as it reflects the scope of the entire project.*

## 1.6 Challenges of Scrum for Data-Intensive Projects

Scrum was born to address the shortcomings of the traditional, plan-driven approach to managing projects. While many of the pain points are shared between software and data-driven undertakings, engineering and science are inherently different. Engineering is precise, and science is uncertain. In software engineering, tasks can be relatively easily identified, estimated, and their progress monitored. Data-driven projects put a great deal of attention on discovery and involve iterative experiments. Learnings from one experiment inform the next one. There is no clear-cut moment when the cycle ends, and the work is oriented toward an outcome or quality threshold. This has two important implications for Scrum in data-intensive projects.

A key challenge is estimation, as the amount of time a given experiment can take is uncertain. If a team is unable to estimate tasks accurately, it cannot reliably decide what can fit into a sprint it in terms of workload. Consequently, this sets a very shaky foundation for the whole framework, which is based on time-boxed and incremental delivery.

Another issue with fixed-length sprints is making very small or very large experiments fit neatly into a couple weeks’ timeframe. The former can result in having multiple, unrelated experiments put into a sprint just to fill it up. What is more, in such a case, a client’s feedback is unnecessarily delayed. Conversely, whenever a complex model is built, some of the underlying activities might require time that extends beyond a sprint. Think of data collection or having a model run long enough to assess its performance.

To address these challenges, new methods to manage data-intensive projects have been introduced. One of them gaining adoption is called Data-driven Scrum (Saltz, 2022). It differs from traditional Scrum in the following regards:

* Capability instead of time-based iterations: An iteration’s duration is decided based on an experiment that needs to be performed. This implies that iterations can have a varying length to accommodate both small and large, complex data-intensive endeavors. The idea is to shift from defining the work that can be done in a specific amount of time to allowing for logical, coherent analytics work to take as much time as needed.
* Provide high-level, rather than detailed estimations: Given the exploratory nature of data-intensive projects and their inherent uncertainty, generating detailed but inaccurate task estimations is not desired. Instead, a high-level scale is used to give an approximation of complexity. Clothing sizes from S to XL can be used when prioritizing the potential tasks.
* Decouple meetings from iterations: Given that iterations can have varying durations, the way sprint reviews and retrospectives are organized needs to be adjusted. It could make sense to carry out a demonstration of multiple experiments at the same time to compare their results and have the same audience present. Retrospectives could move to a calendar-based frequency that makes the most sense for a team, e.g., once per month. In short, retrospectives and sprint reviews are no longer strictly linked to the end of every iteration but can be carried out at appropriate moments in the project lifecycle.

### Self-Check Questions

1. Why are fixed-length Sprints a poor fit for data-intensive projects?

*It stems from difficulty of accurately estimating time needed to execute an experiment and the varying sizes of possible experiments.*

## Summary

In order to increase the likelihood of a successful project completion (i.e., on time and within the foreseen budget), a structured approach to management is needed. Many methods in that area have been proposed over the decades as the technology industry evolved. First, so-called classical approaches emerged where the focus was placed on upfront planning and progressing though sequential project phases. CRISP-DM is considered a traditional approach but is still prevalent in the industry as it provides a generic, well-detailed framework for data-intensive projects. It consists of six phases: business understanding, data understanding, data preparation, modeling, evaluation, and deployment. Each of these comprises a handful of more-granular activities to be carried out by the project team.

While traditional approaches to project management certainly have merits, their focus on documentation and delayed testing of a solution were considered inappropriate by many practitioners. To address such shortcomings, agile methods were devised. They put customer collaboration and short feedback loops at the center and rely on multidisciplinary, self-organizing teams to deliver business value. Scrum is the most prominent framework from the agile family and, like others, it is an iterative and incremental approach. This means that a team works in time-boxed intervals (sprints), each of which ends with a potentially shippable part of the solution. It introduces multiple activities that foster collaboration and continuous improvement: the stand-up meeting, the sprint review, and the retrospective.

The Scrum framework has multiple strengths that make it a good fit for data-intensive projects. It promotes transparency and early testing so intermediate results are shared, client feedback received rapidly, and the tunnel effect avoided. It makes inspection a focal point of the process, in turn fostering flexibility and adaptation based on empirical evidence – simply put, see what works for the team and what doesn’t, and adjust the plan accordingly.

While Scrum has been widely successful in the software industry, it exhibited some limitations when applied to data-intensive projects, which are less deterministic and more difficult to plan. Consequently, some variations of the method started emerging, such as Data-driven Scrum. This moves away from “time-boxed” sprints to iterations of varying duration so that coherent data experiments can be carried out.

# Unit 2 – Infrastructure as Code

**Study Goals**

On completion of this unit, you will be able to …

… discuss the role of infrastructure in data-based solutions and categorize its constituent elements.

… provision basic infrastructure using Terraform and ARM templates.

… explain the added value of using Cloud computing and DevOps in modern data-intensive projects.

# 2. Infrastructure as Code

## Introduction

Infrastructure in everyday language is understood as: “the basic equipment and structures (such as roads and bridges) that are needed for a country, region, or organization to function properly.” In the context of data-intensive projects, that translates into a stack of equipment and resources needed for data-driven solutions to function properly. From this definition, you can already tell that infrastructure is only a means to an end, and not an end in itself.

The ultimate goal is to make data-intensive projects — as well as the teams that are in charge — more successful. Therefore, building reliable and scalable infrastructure sets a foundation to unlocking the “four Vs” of big data:

1. **Volume:** By enabling parallel execution of projects, their total number (hence volume) can increase.
2. **Velocity:** By reducing human errors, decrease the time to market.
3. **Validity:** By programmatically ensuring correct results.
4. **Variety:** By reproductible customization of infrastructure, a larger variety of projects can be supported.

The achieve this, the project delivery process should be automated to the greatest extent possible. That implies coding infrastructure-related tasks rather than clicking around a web interface or manually executing shell commands. This is the concept of *infrastructure as code*.

In this chapter you’ll learn how different infrastructure-related tasks can be achieved programmatically to generate reproductible configurations of data-driven solutions. These are the underpinnings of resilient processes and effective management of technical resources. Before discussing the operational side of using infrastructure as code to automatically provision and handle resources, let’s take a look of what constitutes infrastructure in a typical data-driven solution.

## 2.1 Infrastructure for Data-Intensive Projects

While there isn’t only one way to build infrastructure for data-intensive projects, usually it consists of multiple elements that are put in place regardless of specific project or organizational needs. The figure below presents building blocks of a customary stack of infrastructure for projects whose main concern is data.

A Typical Infrastructure Stack of a Data-Intensive Project

Text

Description automatically generated

Source: Rafal Wlodarski (2023)

### Data Warehouse

A data warehouse is a place where data for your project is stored and which you query to retrieve input for your applications. It is a central repository that integrates data from multiple sources and usually stores both current and historical data. If you deal with a small amount of data, this infrastructure element will likely be represented by a conventional SQL database.

### Compute Resources

As the first layer of infrastructure is solely responsible for data storage, a way to process it and get actual results is needed. Therefore, different types of computations —applying an algorithm, data transformations, model training — are represented in source code or a workflow and executed to turn raw data into business insights. The operations performed on data can vary greatly, but all of them need computing resources like CPUs (central processing units) or GPUs (graphics processing units), which are encompassed in processors. So, a processor is what interprets and executes the instructions of your data-driven solution. In enterprise-level projects, *servers* (units of computation located physically or on a network)are used for this purpose. Whenever you deal with large-scale data processing, tools that are optimized for the job, such as Spark or Hadoop, might be needed.

### Job Scheduler

In data-driven solutions, computation operations are usually performed repeatedly – models are re-trained with fresh data and predictions generated over and over. It is the job of scheduling infrastructure to keep this cycle going. A job scheduler helps to structure and orchestrate different operations or applications that constitute part of a data pipeline. It executes interrelated steps of computation that can be performed at a desired cadence.

### Security Infrastructure

Although it is not a core concern for data-intensive projects, security plays an ever increasing role in such endeavors. Data privacy, access control, and policy enforcement are just some mechanisms that might be required to put in place a consistent security infrastructure. Cloud providers offer specialized resources for identity and access management, as well as logging and auditing, which make it easier to build data-driven solutions that are compliant with general security best practices and/or company standards.

Putting in place, evolving, and maintaining an infrastructure stack to support a data-intensive project throughout its life cycle can be a daunting task. Therefore, it’s important to limit the underlying complexity by reusing existing components instead of setting up resources from scratch. Cloud computing enables this approach by allowing anyone to provision and manage all types of infrastructure in a cost-effective way. Catalogues of all cloud providers include solutions for data storage (Amazon S3), scalable compute resources (Amazon Elastic Compute Cloud – EC2), security (IAM), and finally, higher-level products for data-driven solutions (Azure Machine Learning Studio and Amazon SageMaker).

In order to integrate such elements in your system, you need to provision them first – this means requesting underlying resources with a given cloud provider. This is where tools such as Terraform and ARM templates come in handy – they allow you to employ infrastructure as code and manage your project infrastructure programmatically.

### Self-Check Questions

1. Are certain elements of infrastructure always needed as part of data-intensive projects? If so, which ones?

*Data and computing resources – without the possibility to store (data resources) and process (computing resources) data, no business value can be generated.*

## 2.2 Principles of Infrastructure as Code

Infrastructure as code (IaC) is an IT infrastructure management process that enables automated provisioning and handling of the technical resources necessary to run a data-driven or software solution. This encompasses anything from setting up servers, virtual machines, databases, and networks to application configuration and deployment steps.

The idea behind IaC is to write a (most commonly text) file containing code to define, deploy, update, and destroy a project’s infrastructure. This is then executed and takes care of the system administration without human intervention. Before IaC was born, there were dedicated operations teams performing such tasks manually. This represents an important technological and organizational shift where handling of hardware is done in a programmatic, so software-oriented way.

However, such a change can be a difficult exercise fraught with many challenges. Principles of IaC can help a team overcome potential pitfalls and make the most of this modern IT infrastructure management process.

### Systems can be Easily Reproduced

Any element of a project’s infrastructure should be able to be rebuilt reliably without needing any important decision-making. The first element that this implies is that there should be very little (or no) risk attached to the process of introducing changes so failures can be addressed quickly. The second constituent of the principle in practice means that any operational information – think versions to install on a server, a hostname, database connectivity etc. – shall be included in the scripts or tooling that handle the task. In short, new infrastructure should be provisioned with as little effort as possible.

### Systems are Disposable

Infrastructure as code enables automatic and efficient infrastructure management by programmatically executing tasks like resource creation, deletion, replacement, etc. To make the most of this approach, it needs to be accompanied by a shift in thinking at the design level as well. The idea is that systems should be assumed to have constantly changing infrastructure, and that this does not impact their operations. A model should continue classifying data even when the underlying servers are resized or replaced.

### Systems are Consistent

To trust your automation process (see the first principle), consistency must be ensured at the infrastructure and configuration levels. For example, imagine two infrastructure elements providing the same service – for example, application servers that ensure the scalability of a data-driven project. For your team to use them reliably, they should be nearly identical; otherwise the solution might behave differently for different end users.

At a configuration level, this requires the team programming the underlying resources to handle any planned differences in a particular manner. They should either apply the modification to all concerned infrastructure elements or introduce a new class or role for a subset of servers that will have different configuration. In this way, a server of a specific type can be provisioned in a consistent way every time the operation is repeated.

### Processes are Repeatable

As a natural next step of the reproducibility principle, any infrastructure-related task should be repeatable. That’s how things work by definition in automation tools, and the problematic part is human nature. If a seasoned administrator is faced with a one-off task, it might be easier to execute it manually. However, a strong scripting culture is at the core of any effective infrastructure team. Therefore, even if it might not be the fastest way to do things, if a task can be scripted, it should be scripted. Whenever an operation is not obvious to script, investigate if there is a technique or tool that allows it to be performed programmatically.

### Design is Always Changing

Early approaches to software development relied on the assumption that systems are difficult and expensive to change. A major shift in thinking was represented by agile approaches, which embraced evolving requirements. Things were revolutionized again with the advent of cloud computing. With IaC, making changes to the resources of a system is easy and cheap. Having said that, fostering change at the system’s design level is necessary. Data-driven solutions and their underlying infrastructure must be conceived in a straightforward manner, avoiding any complexity that might make changes more difficult.

Another way to ensure that a system can be modified safely and quickly is by incorporating changes frequently. This way, the team enforces good habits and learns how to manages changes in a streamlined fashion.

### Self-Check Questions

1. With Infrastructure As Code, should every task be scripted by default? Why so?

*Yes, this stems from the “Processes are repeatable” principle. If something is not scripted it cannot be easily repeated.*

## 2.3 Terraform

Terraform is one of the leading IaC tools, used for resource provisioning and management. It allows you to define the target infrastructure elements programmatically (data storage, computing, and security resources) and will automatically request them with a given cloud provider.

Terraform is open source, which implies there is no link between the automation tool and any cloud company. This means that you can use it to manage your project’s infrastructure independently of who provides the underlying resources. In the next section, you’ll get familiar with another tool for which that is not the case – ARM templates.

**Application Programing Interface (API)**

This is a set of procedures and specifications that enable interactions between programs. API calls are realized by message exchanges between a requester and provider to perform some action defined by a programmer.

**Binary file**

A file whose content can be interpreted by computers (as opposed to a text file, which is human-readable).

### How Does Terraform Work?

Infrastructure as code written in Terraform compiles into a single binary file, called *terraform*. Such a file can be then used to deploy infrastructure directly from your laptop or a solution server or any other machine. This is possible as the file makes API calls to infrastructure providers such as AWS, Azure, Oracle, and others.

To define where a Terraform file should make the API calls (hence manage the infrastructure), a Terraform configuration file needs to be defined. This is a text file that corresponds to the “code” part of “infrastructure as code.” This is where you “program” your infrastructure by defining commands that respect Terraform’s syntax (rules that define the combinations of symbols that are considered to be correct). Let’s look at a very simple Terraform configuration file.

output "hello\_world" {

value = "Hello, World!"

}

Even without prior programming or infrastructure management experience, you can tell that it will display the text “Hello, World!”. But you must be wondering where? Somewhere on Azure or Google Cloud? Not really, we’re not there yet. It will display the text wherever the file is run. So, if you would like to test it and see it in a command line on your computer, you need to [**install Terraform locally first**](https://learn.hashicorp.com/tutorials/terraform/install-cli?in=terraform/).

Of course, on a commercial, large-scale project, the process would look a little different. Someone would define entire project’s infrastructure – servers, databases, and so on – in Terraform configuration files and store them in a central code repository. They would then be run with Terraform commands, such as terraform apply, to deploy the infrastructure defined in the files. The workflow is very simple: create Terraform configuration file(s), commit them to a source code control repository, and run them with Terraform commands (this would normally be done on a CI/CD server) to manage your infrastructure whenever you need to.

By executing the commands, your human readable file is translated to a binary form, parsed by a machine, and translated into a series of API calls to the cloud providers. A simplified view of the infrastructure management process with Terraform is depicted below.

A Simplified Process of Infrastructure Management with IAC on a Large-Scale Project

Diagram

Description automatically generated  
Source: Rafal Wlodarski (2023)

Whenever a change is needed to the infrastructure, instead of updating the resources manually on the servers, a team member performs their modifications to the Terraform configuration files. Validation takes place though automated tests; changes are committed to the project central repository and then run via the terraform apply command.

Let’s look at the mechanisms and programming notions that allow one to manage infrastructure in Terraform.

### Setting up Your Environment

Terraform can provision infrastructure from a wide range of cloud providers, including Amazon Web Services (AWS), Azure, Google Cloud, and OpenStack. The code samples that you’ll see in this chapter will use AWS. It was chosen for two main reasons. It’s the most popular cloud infrastructure provider, so chances are you will use it at some point if your career. Secondly, AWS offers a generous limit of free resources that you can use to run the examples provided and tweak them further.

In order to set up a AWS account, follow [this tutorial](https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/).

This chapter requires you to have Terraform installed locally as well. Terraform will need to make changes in your AWS account. To enable this, you need to set the AWS credentials for the IAM user you created beforehand. See the snippet below for an example in a Unix/Linux/macOS terminal:

$ export AWS\_ACCESS\_KEY\_ID=(your access key id)

$ export AWS\_SECRET\_ACCESS\_KEY=(your secret access key)

These environment variables apply only to the current session, so if you reboot your machine or open a new terminal window, the operation needs to be repeated.

### Basic Example: Deploying a Single Virtual Server with AWS

You will “code” the infrastructure needed in text files with **.tf extensions** by declaring whatever resources or operations you need. Terraform will take it from there and translate your wishes into actual infrastructure (making API calls to cloud providers as an intermediary step).

You can write Terraform code in any text editor (and use the “save as...” menu to specify the .tf extension) or a tool that supports Terraform (or HCL) syntax highlighting. This will make it easier for you to specify correct commands; examples of such tools include vim and Emacs (simple) or Visual Studio Code and IntelliJ (more advanced).

The first step to manage your infrastructure is to identify and configure the provider(s) you wish to use. As mentioned, we will employ AWS. It has many data centers grouped into regions, so you need to specify the one where you want to deploy your infrastructure. To do so, prepare an empty folder and a created a file called main.tf inside it. The file should look like the following:

provider "aws" {

region = "eu-central-1"

}

The next logical step would be specifying the resources you want to create. The following general syntax allows you to do so in Terraform:

resource "<**PROVIDER**>\_<**TYPE**>" "<**NAME**>" {

[**CONFIG** ...]

}

The parameters you need to indicate are pretty intuitive: PROVIDER (AWS in our case), TYPE of resource (database, server, security group), NAME you’ll use to identify and refer to it, and CONFIG, which comprises one or more arguments that apply to the resource in question.

Let’s try to deploy a single (virtual) server in AWS, known as an EC2 Instance. Write a resource command in main.tf as follows:

resource "aws\_instance" "example" {

ami = "ami-0ee23bfc74a881de5"

instance\_type = "t2.micro"

}

The CONFIG part contains the minimum required arguments to create an AWS server. There are a lot of parameters possible, and you don’t need to understand them all. The required ones are as following:

* ***ami***: the Amazon Machine Image provides information on the operating system to be installed on your EC2 instance. There is a variety of free and paid AMIs available on the AWS Marketplace website (under operating systems category, delivery method: Amazon Machine Image). In the example an Ubuntu bionic 18.04 version is used (a Linux-based operating system).
* ***Instance\_type***: a virtual server, like any other server, has various characteristics, including CPU, memory, disk space, etc. The type of a specific EC2 instance defines such configuration, and you can simply choose one of the available versions on the AWS website (while keeping in mind that the more powerful your machine is, the more costly it most likely will be).

#### Use of tags

For AWS resources, it is possible to associate **tags** to provide whatever metadata could be useful, e.g., for auditing purposes. Tags are simply key–value labels you can assign to AWS resources that give extra information about it. The code snippet below includes only “Name”, which is a human-readable resource name, but other tags are commonly used as well: environment, project, owner, app-id (the application using the resource), and app-role (the resource’s technical function: server, database, etc.). Let’s refactor slightly our terraform configuration file to associate a name with the EC2 instance we aim to create.

resource "aws\_instance" "example" {

ami = "ami-0ee23bfc74a881de5"

instance\_type = "t2.micro"

tags = {

Name = “EC2-example”

}

}

With the terraform configuration file clear and ready, all that is left is letting Terraform translate the “configuration as code” you developed into API calls to Amazon AWS. As this is the first time you are doing it, there are a couple of initial commands to invoke.

**Refactoring**

The process of modifying existing code to improve its structure, design, or any other aspect without changing the underlying behavior.

#### Terraform commands to run your IAC

In your shell terminal, go to the folder where the main.tf file was created and run the terraform init command. Upon completion, a message “*Terraform has been successfully initialized!*” should be displayed. What happened is that Terraform scanned your configuration file to identify the cloud provider of choice and downloaded (into a *.terraform* folder) code to be able to handle it. You just need to remember to run the terraform init command whenever you start with a new configuration file.

The next step is to run the terraform plan command, which will display the actions that Terraform will take (in our example, “*aws\_instance.example will be created*”). On the bottom you should see a summary of all infrastructure changes you plan (add: resources preceded by a “+” sign, change: “~”, destroy: “-”). Performing such a sanity check allows you to verify that the modifications are as desired; to perform them, run the terraform apply command.

Congratulations, you have deployed a virtual server on the biggest cloud provider using the “infrastructure as code” process! You can see the fruits of your labor on the EC2 console; something similar to the content of the screenshot below should appear. The “Name” column should correspond to whichever value you associated with the name tag.

A Deployed Instance of EC2 Virtual Server as Seen in EC2 Console (AWS Website)

Graphical user interface, text, application, email

Description automatically generated

### Useful Example: Deploying a Configurable Web Server with AWS

In this section, you’ll “code” infrastructure that you can actually use for your data-intensive projects: a web server that can respond to HTTP requests. A couple of Terraform mechanisms/techniques will be introduced as we progress with the example – they will always be preceded with a title in bold. Keep this in mind, as such techniques can be reused in your future endeavors.

In order to enable incoming and outgoing traffic from an EC2 instance (by default this is not the case), a security group needs to be set up. It is a required, security (you guessed it!) mechanism that acts like a virtual firewall that controls access to your AWS resources. The following code snippet creates a new security group that allows incoming (defined in *ingress* section) TCP requests on port 8080 (standard port for such activity) from any IP address range (0.0.0.0/0). The last element is specified using the CIDR blocks notation, which is one of the methods to define a range of IP addresses (you can read more on virtual private clouds and CIDR blocks in AWS’s documentation (Amazon, 2022c)).

resource “aws\_security\_group” “instance” {

name = “terraform-sample-instance”

ingress {

from\_port = 8080

to\_port = 8080

protocol = “tcp”

cidr\_blocks = [“0.0.0.0/0”]

}

}

With the security group set up, you need to indicate to your EC2 instance that you will use it. This is achieved by passing the security group’s ID into the vpc\_security\_group\_ids argument of your server. The ID in question is not a static value (it will be assigned once the underlying resource is created), so you cannot “hardcode” it in your configuration.

#### Using Terraform expressions

**Network Security**

For simplicity of the examples, the infrastructure is deployed to public IP address space (this also allows you to test the configuration from your computer). For production systems and data, this is considered a security vulnerability, and you should use private IP space instead.  
For more information, refer to the AWS documentation (Amazon, 2022a,b).

Instead, you need to employ a **Terraform expression**. This is a mechanism that returns a value – be it simple text or, in our case, a reference pointing to another point in the code. In order to retrieve the ID of the security group created, the following syntax will be used:

<PROVIDER>\_<TYPE>.<NAME>.<ATTRIBUTE>,

where similarly to our previous step, PROVIDER is the name of the cloud provider (AWS in our case), TYPE indicates the resource type (server, security group), and NAME identifies a particular resource. ATTRIBUTE can be either an argument of the resource in question or one of its available attributes (different fields can be used depending on the resource type – you need to refer to AWS’ documentation for a list of attributes). We are interested in the security group’s ID, so the **expression** takes the following form:

aws\_security\_group.instance.id

We can plug it into our code in square brackets to indicate that we are working with an attribute. The enriched EC2 instance declaration below contains the information on the security group it will use:

resource "aws\_instance" "example" {

ami = "ami-0ee23bfc74a881de5"

instance\_type = "t2.micro"

vpc\_security\_group\_ids = [aws\_security\_group.instance.id]

tags = {

Name = “EC2-example”

}

}

What happens here is we provide a reference to a value associated with a resource that was declared earlier (aws\_security\_group, in the previous code snippet). That resource will have an ID allocated once it’s put in place by AWS.

**Terraform resource creation order**

You might have noticed that we go back and forth to our different resources and modify the code. Did you wonder if these resources need to be placed in a specific order for everything to work correctly? The answer is no. Terraform analyses expressions within a resource block to find references to other objects, and figures out the ordering of creating, updating, or destroying resources.

This mechanism allows us to dynamically plug values into our infrastructure. It’s an important ability as sometimes you simply might not know a given value upfront so you can’t hard-code it.

#### Using variables

Another scenario in which you need some sort of reuse mechanism is whenever a value appears multiple times in your code. Most commonly, instead of typing the value over and over, you would declare a **variable** and refer to itin all the places its value is needed. You can think of a *variable as a container that stores a value of your choice*, which you can overwrite if need be.

A keen eye might have noticed that in our example the value “8080” – which represents a connection port – appears twice. Let’s refactor our security group configuration code using Terraform variables. This approach improves the maintainability of our code – if you wish to change the port used (which is likely to happen), you don’t need to change it in multiple places but modify the variable value instead. Having to manually refactor code in different places is a very error-prone process that IaC aims to address.

The syntax of variable definition is similar to the one for declaring resources along with their configuration – but we use the keyword **variable** instead. Variables have a type associated with them, which indicates what kind of value will be stored inside (e.g. numeric, string to represent characters, or Boolean to indicate logical values). The actual value is specified with the **default** parameter. See the code snippet below for a full example:

variable "server\_port" {

description = "The port number to handle the server’s HTTP requests"

type = number

default = 8080

}

To use the defined variable in our code, we need to provide a *reference* to it with the following syntax: var.<VARIABLE\_NAME>. Terraform will look up the variable with the name you provided and retrieve its value. Therefore, our refactored security group resource definition looks like the following:

**Terraform modules**

To make your infrastructure as code even more reusable and manage resources in a coherent way across multiple environments, you can employ the Terraform modules mechanism. Please refer to the book Brikman (2019), chapter 4 for more details.

resource “aws\_security\_group” “instance” {

name = “terraform-sample-instance”

ingress {

from\_port = var.server\_port

to\_port = var.server\_port

protocol = “tcp”

cidr\_blocks = [“0.0.0.0/0”]

}

}

Both *expressions* and *variables* allow us to make the infrastructure code *configurable.* This allows to avoid duplication and makes the code easier to reuse – next time you (or your teammate) need a similar configuration, all you have to do is substitute the values in your expressions or variables.

In this section, we used Terraform as an infrastructure as code (IaC) tool to set up a single, configurable Web server. This can be a starting point for making a data-intensive solution available to public or for testing, by running your application on the created EC2 instance. While this is out of the scope of this chapter, you may refer to one of the tutorials to give it a try (Yildirim, 2020; Gallatin, 2021).

#### Resources clean up

Having followed the tutorials and experimented with IaC using Terraform, it is good practice to remove all the created resources. Remember this is not something you would perform in an actual project but will help you limit the consumption of your free tier AWS budget. All there is to do is run the destroy command on your terminal (where you executed other Terraform commands).

As a reminder, please refer to the diagram below that summarizes the sequencing of all the Terraform commands that you’ve learned.

A Typical Terraform Workflow to Manage Your Infrastructure as Code

Source: Rafal Wlodarski (2023)

### Self-Check Questions

1. What does a declarative language mean in the context of IAC?

*It allows to declare a desired state of the infrastructure.*

1. What does the “apply” command do in Terraform? Please mark the correct statement(s).

* *Applies a desired infrastructure to the cloud*
* *Destroys the configuration in the cloud*
* *Modifies the configuration in the cloud*
* *All of the above can be true*.

## 2.4 ARM Templates

You already have a good idea of what infrastructure as code entails and how it can be used to programmatically manage your project’s resources. The choice of where the resources are put in place might depend on many factors (technological, budgetary, privacy, and security among others), but in modern, constantly evolving solutions, cloud is the number one choice.

Terraform, which you got familiar with in the previous section, is a provider-agnostic solution, so it is not bound to a particular cloud services company. This has its pros – such as less effort if you need to move your infrastructure to another cloud provider – as well as cons – not fully leveraging on a specific cloud solution or experiencing some limitations for the same reason. Therefore, some companies decide to leverage a native solution that their cloud provider of choice proposes. That’s exactly what Azure Resource Manager or simply ARM templates is – an IaC tool created by the world’s second biggest cloud company: Microsoft.

The principles behind it and objective are the same as Terraform: It is declarative, so it allows you to specify the desired infrastructure state and repeatedly deploy it via automated processes. Nevertheless, it uses JavaScript Object Notation (JSON) files and has different syntax. What this means in practice is that you can carry out pretty much the same operations as in Terraform, but the keywords, parenthees, commas, and other language elements you use to “code” the infrastructure differ.

Microsoft, being one of the world’s biggest and most successful software companies, takes its business seriously and provides comprehensive documentation and learning materials for ARM templates. Consequently, rather than summarizing them here and risking providing you with incomplete information (and potentially out-of-date in the future), this section points you to the materials that will help you get up to speed with Azure Resource Manager should you need to use it one day.

The overall process is the same as what you already know: Set up your development tool where you’ll program the desired infrastructure, create an Azure account thanks to which the resources will be provisioned, and start creating those configuration files!

It is customarily easiest to begin with a basic end-to-end tutorial, which “Create and deploy your first ARM template” covers (Microsoft learning, 2022a).

As a follow up, you can employ the same mechanisms as in the Terraform section – namely add resources to your configuration file (Microsoft learning, 2022b) as well as adding variables (Microsoft learning, 2022c) (remember code reuse?). If you are more at ease with video tutorials rather than reading materials, Microsoft has created a video learning path on ARM templates basics (Microsoft learning, 2022d).

It is advised that you gain a better understanding of the syntax and structure of ARM templates (Microsoft learning, 2022e). You will then be able to identify relevant sections and properties of an ARM template to programmatically reflect your target infrastructure.

If you would like to re-do the example carried out with Terraform and create a web server on Azure, a quick start document (Microsoft learning, 2022f) will help you with that.

Finally, familiarity with good programming practices will help you create clean and secure infrastructure code (Microsoft learning, 2022g).

### Self-Check Questions

1. Can one use Terraform for Microsoft Azure? If yes, why would you choose it over ARM templates?

*Yes, as it is provider-agnostic.*

*It would make a move to another cloud provider simpler.*

## 2.5 DevOps

Not so long time ago, building software or data-driven products implied managing the hardware needed to run them as well. In practice, that meant dedicated rooms for servers, which needed to be wired and cooled to ensure their proper functioning. Whenever a large amount of such hardware was to be managed, dedicated personnel would take care of that – a so called IT operations team.

Development of software was confined to a separate team of engineers that performed completely different types of tasks, and once a product was ready, they would hand it over to the operations team for hardware set up, deployment of the software (that mostly involved manually running commands on a server), and technical maintenance. Silos formed with little communication between the disparate teams.

While there is nothing wrong with such an approach per se, whenever the company experiences significant growth and products rapidly increase in number (consequently the underlying hardware and infrastructure as well), things get out of hand. Every new software release takes more and more time and becomes less predictable. Given that humans make errors whenever manual work is involved, problems arise more often, which leads to less frequent releases and longer time to market. Teams grow unhappy and so do customers, who do not obtain their bug fixes in a timely manner.

To address one of the underlying problems – managing infrastructure in-house – plenty of companies seized the opportunities created with the arrival of cloud computing. Cloud providers like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) offer on-demand, scalable infrastructure-related services so that companies don’t have to manage their own datacenters, servers, and the like. As a consequence, operations teams shifted to employing programmatic tools for managing cloud infrastructure and no longer needed to set up and maintain physical, on-premises resources.

With time, both development and operations teams grew less specialized in either side of the project delivery, and the distinction between the two teams became blurred. Responsibility of operational and application code was more evenly distributed thanks to automation and both teams work more closely.

This is what DevOps culture means – having both development and operations teams collaborate rather than work in silos. DevOps is not a tool or a technology but a cultural and organizational dimension of product development and delivery. It relies on automation (infrastructure as code, and continuous integration and delivery, more precisely), short feedback loops, and release cycles to build and ship software and data-intensive projects faster, more efficiently, and more reliably. One of the success factors of DevOps is its human aspect, which manifests itself as close collaboration within the team and fostering multi-disciplinary skills.

### DevOps Lifecycle

DevOps impacts the whole life span of a product: from development, through delivery and maintenance. Due to is continuous nature, the DevOps lifecycle is often depicted as an infinite loop (see figure below), which is composed of eight phases. They represent the processes, capabilities, and tools that both the development (on the left side of the loop) and operations (on the right side of the loop) teams need for DevOps to succeed. Through each of the underlying phases, teams communicate, collaborate, and seek improvement continuously.

DevOps Lifecycle: Underlying Phases

Diagram

Description automatically generated

Source: Rafal Wlodarski, based on Atlassian, based on: <https://www.atlassian.com/devops>

**Discover**: This is when gathering inputs and feedback from key stakeholders takes place. This highly interactive phase seeks to understand the current processes and tools by capturing information in templates and checklists. Such input allows a list of DevOps methods and practices to be compiled that address existing challenges and constraints.

**Plan**: For the team to be effective, future work needs to be identified, prioritized, and then its progress tracked. For each of these activities, an agile way of working should be applied and carried out according to one of the associated methods or frameworks, e.g. Scrum.

**Build**: This is when the actual solution is developed, so common activities for a data-intensive project are data understanding, preparation, modeling, and evaluation. In terms of technical processes, this phase relies heavily on source code management and continuous integration.

**Test**: The DevOps test strategy is about “shifting-left,” which simply put means testing early in the project lifecycle. Tests should be written at the lowest level possible and run automatically before any new feature is integrated into the code repository.

**Deploy**: With DevOps, the application code moves through the release pipeline to the production environment continuously. Before reaching end users, each project increment typically involves a successful continuous integration build, followed by automated tests of different levels and executed in multiple test environments.

**Operate**: After a successful delivery of the solution, operations activities encompass maintaining, monitoring, and troubleshooting it in the production environment as well as managing all IT infrastructure.

**Observe**: DevOps focuses on observing the impact of the solution delivered on its users and infrastructure. This requires collecting appropriate data so that product uptime, speed, and functionality can be safeguarded.

**Continuous feedback**: DevOps teams need to evaluate each release so that they can improve in the future. Incorporating customer feedback in terms of the solution and its development and delivery process is a key aspect.

As DevOps implies a multitude of tools and methods, as well as a cultural shift, it might be very challenging to roll it out all-at-once for the whole organization. Therefore, to get started it is advised to start small – identify a supporting application or service where the team could experiment with this new way of working. Adopting DevOps culture, practices, and tools gradually will help increase the team’s confidence and ultimately achieve business goals faster.

### Self-Check Questions

1. What is the overlap between activities that make part of the DevOps lifecycle and “traditional” operations team’s responsibilities?

*Operations team would usually be involved with all the activities that start with deployment and any operational efforts that follows(hence deploy, operate, and observe phases).*

### Summary

Long gone are the days when operations teams’ main responsibility was setting up servers, rolling out new products to production, and maintaining the related infrastructure. This manual, error prone process was replaced with a programmatic approach that is repeatable and reliable — so called *infrastructure as code*. Many tools that enable it (Terraform, ARM templates) work in a similar fashion — one declares the desired infrastructure in a human-readable file, and the tool itself takes care of the rest.

This was made possible by the arrival of cloud computing. The way infrastructure is provisioned has changed dramatically. Companies no longer need to own servers, databases and the like but can have resources available on-demand from cloud providers. While the set up can be performed via consoles and web interfaces provided by the cloud companies, this is usually done directly by the infrastructure as code tooling. After all, the idea is to automate as much of a product lifecycle as possible.

That is the premise of DevOps — automate and integrate the processes across product development, delivery, and operations. DevOps is a portmanteau of development and operations (teams) and represents a combination of culture, practices, and tools that enable faster and more streamlined delivery of business value.

# Unit 3 – Testing and collaboration

**Study Goals**

On completion of this unit, you will be able to …

… discuss technical building blocks and processes underpinning collaboration on data-intensive projects.

… compare different branching strategies, choose the relevant one for your project context and demonstrate its workflow.

… differentiate project environments and discuss their particularities.

… recognize varying approaches to testing.

… apply and evaluate high-to-mid granularity tests of a data-driven solution.

## Introduction

It is rarely the case that one data scientist conceives, implements, tests, and delivers a full-scale solution for commercial use. Such endeavors are usually a collective, long-term effort of a diverse team. Furthermore, team members working on a data-driven solution should not do so in isolation but instead share their deliverables early for others to participate in their development, review, verification, and validation. While a collaborative approach to building data-driven solutions is a given, certain building blocks need to be put in place on every project to avoid work sharing and synchronization issues.

One of them is a repository – a centralized archive to store and version all project artifacts. This is where the development team share their efforts and where the latest version of the source code can be found. In practice, to avoid conflicts and overwriting others’ work, a well-defined approach to management of a repository is needed and must be respected by the whole team.

Building a data-driven solution represents only one facet of collaboration. Ensuring that whatever changes were made result in a working piece of the final product is another part of the story. That typically requires both internal project and client teams to execute automated and manual tests on the solution under development. Again, work synchronization issues are likely here, so dedicated project environments are used for various purposes.

You can already tell that whenever multiple people are involved in a project, collaboration challenges arise. To anticipate and address them, well-defined approaches and guidelines are needed. This chapter deals with modern collaboration and testing practices that enable teams to efficiently produce data-driven solutions that run smoothly and meet client expectations. You must be wondering, what is the key to success? There is no silver bullet solution, and different tools and processes might be employed depending on your project context. Let’s find out!

This unit elaborates on multiple topics, and some of them are building blocks for others. The first section discusses basic notions of repository management and branching strategies that can be applied on a project.

Next, commonly used environment types for data-intensive projects are presented. These are a direct outcome of repository management activities and serve differing purposes for the development team and the client. These varied needs can be linked to project lifecycle phases and are discussed from the perspective of testing stages in the second section of the unit.

Finally, different approaches to testing activities are described and a classification of test types and levels provided. They represent the underpinnings of testing stages discussed earlier.

## 3.1 Branching Strategies

Modern technology projects rely heavily on collaboration between team members who oftentimes work on the same artifact in parallel. To enable effective, conflict-free coordination of work, some underpinning mechanisms and tools are necessary. First, a (code) **repository**, which is simply an archive containing all the information needed to manage the revisions and history of a project. Think of it as a remote database that stores a complete copy of your entire project – that includes its structure, its metadata, and the underlying files. To use a repository efficiently, a tool that manages and tracks different versions of a solution is needed. This is referred to as a **version control system** (VCS), and a ubiquitous example of such a tool is called Git. While the basics of repository management with Git are out of scope of this coursebook, you can familiarize yourself with the underlying mechanisms and concepts in Loeliger (2012) and Anstey (2020).

When working with a VCS, different **branches** are created under the hood. They are areas of a repository that represent a specific version of a project. A branch enables launching a separate line of development on a project. That makes concurrent development of the same solution possible, and every team member can work on new features, fix bugs, or safely experiment with new ideas without impacting the efforts of others. Often at some point in the development, branches are reconciled to reunite artifacts produced by multiple team members and create a common version of a solution.

There are a handful of types of branches that are used on technology projects (data-driven solutions are no different in this regard) and which correspond to re-occurring needs in such endeavors:

* **trunk branch** (also referred to as *main*/master branch or mainline): This is the default, automatically created first branch in any repository.
* **development branch**: This is a generic, long-lived branch where work of developers is stored before it reaches a state that’s ready for release. It often parallels the main branch and is never removed.
* **feature branch**: This is created to isolate the development of a well-defined or conceptually separate task or functionality. It can be short- or long-lived and shared among team members or dedicated to just one of them – all depends on the branching strategy used.
* **release branch**: This is used to hold changes that represent an individual customer release so are meant to be delivered to a production environment.
* **hotfix branch**: This is used to isolate changes that address an emergency issue. It is based on the trunk branch and represents a bug fix that needs to be promptly released to production.

Let’s take a look at an example. Imagine you are working on a system that includes three pipelines: one for feature generation, one for training, and an inference pipeline. The simplest approach would be to keep all the corresponding code that is production-ready in one branch (trunk) and implement and maintain the solution in a dedicated development branch.

However, the organization of this system maps naturally to features (each pipeline representing one: feature generation, training, inference). Therefore, there could be one feature branch for every pipeline where dedicated team members work independently of other sub teams. In such a case, there would be a need for another branch to keep all the code that integrates work from feature branches – e.g., the trunk branch. Once an appropriate level of quality is achieved for the common code base, it could be moved to another branch that represents a production-ready version of the solution – e.g., a release branch.

Both approaches to repository organization could account for special cases where the solution delivered to the client requires some urgent maintenance work (e.g., a critical bug in the inference pipeline stopping it from producing any results). In order not to disrupt the ongoing development work, a hotfix branch (based on the source code version where the issue was detected) could be created, the bug addressed, and the new code promptly released to the client.

A branching strategy builds on the terms you’ve learned so far. It refers to an approach that a team employs when writing, merging, and shipping code via a VCS. It is a process that’s meant to ensure every person involved in the development makes changes to the repository in a consistent way. On the ground, a branching strategy is the recommended workflow – a sequence of steps – therefore all the team members need to respect this to accomplish work in a productive manner. To sum up, a branching strategy uses different branch types for well-defined purposes (like in the example) but on top of that prescribes a particular order of tasks that is expected on them.

There are multiple branching strategies for different project contexts; consequently some questions need to be addressed when deciding on which one to choose. Is the underlying workflow easy to learn? Does the approach scale for large teams? Is it easy to revert changes? Don’t worry, you don’t have all the answers yet. Furthermore, a branching strategy can be changed during a project lifecycle if it turns out to be a poor fit with the team.

### Centralized Workflow

As the name suggests (sometimes it is also called trunk-based development), this uses a centralized repository to integrate all changes. Therefore, the main branch is a single point-of-entry for any developer’s work. No other branches are needed in this branching strategy.

A team member joining a project starts by cloning the repository and keeps a local copy of the solution. He/she edits/adds files as needed and commits the changes to the main branch. The golden rule of trunk-based development is that changes need to be integrated to the repository at least once per day.

Imagine a scenario where Adam works on his feature for some time employing the standard Git commit process – edit, stage, commit. As these commands result in local commits, there are no impacts on the central repository. Now, Betty has been developing another functionality in parallel in her own local repository. After two weeks they simultaneously decide (while being unaware of their respective developments) to publish the local changes and integrate their work against the common code base. What they don’t know is that both have modified the same line of code. On top of this, Adam altered another line, which Betty actually deleted. These parallel changes to the same line of code generate a merge conflict. See the diagram below for a simplified visualization of the aforementioned scenario (source code lines marked in color constitute the mere conflict).

Merge Conflict in a Centralized Workflow

Diagram

Description automatically generated

Rafal Wlodarski (2023)

When an algorithm tries to combine such changes, it often results in a merge conflict. When there are more files to integrate it can get messy.

Sounds familiar? When working on past programming assignments, you’ve probably held your changes locally until right before the deadline and had to spend hours integrating them with your peers’ work.

The daily commit rule of trunk-based development was formed due to this recurring pain point. It forces the whole team to regularly reconcile their work, avoiding potential conflicts.

A major shortcoming of the centralized workflow is the conflict resolution process. While the scenario above involved only two people, it can easily become a bottleneck as the size of a team grows. You must be wondering why this branching strategy is popular?

#### When to use the centralized branching strategy

One of its major selling points is simplicity. As discussed, it does not require any other branch to be put in place or used apart from *main*. Hence the centralized workflow is easy to understand and follow. This makes it a good fit for three common scenarios:

1. **When working alone**: If the project you work on is a one-person show, there is no need to complicate your life and manage multiple branches.
2. **When working on a small team**: It may be the case that project team members creating a data-driven solution have separate areas of expertise and are not numerous. Hence, they work on distinct parts of the system, possibly even using different technologies, and the risk of having a merge conflict is very low.
3. **When aiming for high speed of delivery**: If done right, trunk-based development can be the fastest way to get changes into production. It does not impose a time-consuming code-review practice (more on that in a moment) and avoids the high overhead associated with other branching strategies.

### Feature-Branch Workflow

With this branching strategy, *main* represents the official version of the solution that can be released to the client. From it spawn **feature** branches that are created for a dedicated part of a system that represents a separate, logical entity. The idea behind this is that a team member or sub-team can work on independent features without impacting the main branch.

As from the technical (Git’s) point of view both types of branches are the same, feature branches need to have descriptive names, depicting what a given feature does – e.g., linear-regression-advertising-revenue-prediction or outlier-issue-#24. A feature name should convey a clear and highly focused purpose for the branch.

A typical workflow of the feature branching strategy (that each team member should follow) consist of multiple steps:

1. Create a new branch from the latest version of the main branch.
2. Check out the branch locally to ensure any changes will be made to this feature branch.
3. Continue working on the feature on the same branch using the usual Git process: edit, stage, commit.
4. Regularly push the feature branch to the central repository. The purpose of this action is two-fold – create a remote back up (as opposed to the local one) and integrate your changes with the efforts of others should other team members be working on the same feature.
5. Once a feature or its standalone part is ready, open a **pull request**. This step is mandatory before merging your changes into the *main* and creates an opportunity for other team members to **review your code**. The objective is to have a second pair of eyes go through the changes, detect any issues (e.g., logical errors, violations of coding standards, unaddressed functional requirements) and ask for corrections/improvements. On the one hand it ensures quality on the project and on the other it’s a knowledge sharing exercise. If some improvements are requested, the workflow involves additional steps:
   1. Perform any modifications indicated in the pull request. The usual Git process needs to be employed: edit, stage, commit.

**Rebase**

Rebasing is a process of changing the base of a given branch. It implies moving or combining commits into a new base commit. It makes it appear that a branch was created from that commit.

* 1. Push the updates to the central repository.
  2. The pull request is accepted by team member(s).

1. Once the pull request is approved, rebase the feature branch on *master* to ensure the most recent version of the code base is used.
2. The feature branch is merged into the main, thus integrating the changes into the production targeted version of the solution.

Please refer to the diagram below to understand how different branches and the corresponding solution versions intertwine in the feature branch workflow.

Feature Branching Strategy Example With Two Features Developed in ParallelChart

Description automatically generated with low confidence

Rafal Wlodarski (2023)

#### When to use the feature branching strategy

Clearly the complexity of this branching strategy increases dramatically when compared to trunk-based development. Nevertheless, this is a commonly used approach to collaboration on a code base. When should it be considered?

* When multiple features need to be developed in parallel: Sometimes it’s easy to organize a solution around distinct, highly independent features, e.g., based on a business process or functionality. In such a case, development can be done independently on separate feature branches by different team members or sub-teams. This is a very efficient way to organize and scale work.
* When multiple team members need to work on a part of the solution: Having a dedicated feature branch enables easy collaboration with other developers. Collective efforts of the sub-team are integrated on regular basis, and once a feature is ready it can be merged into the main branch.
* When a higher level of code quality control is required: As opening a pull request (and as consequence having a code review) is a mandatory step of the feature workflow, it requires at least one team member to validate any changes before they reach the main branch. This can be useful whenever there are many junior employees working on a project.

### GitFlow

The last branching strategy that is commonly encountered on projects is called GitFlow. It relies on numerous, long-lived branches that only get merged to *main* once the underlying (large) feature is fully complete. Therefore, it is often time-consuming and risky to integrate changes to the main branch as developers work in isolation for long periods of time.

Although GitFlow is historically significant (mostly popular in early 2010s as use of Git spread), nowadays many of the teams opt for the feature-based workflow instead. It uses the same concepts, but branches represent features of high granularity, therefore requiring less time to implement; this translates into less deviation from the main branch and hence less collaboration effort needed to integrate the changes.

GitFlow is adapted to projects that are explicitly versioned, meaning they have a release cycle defined upfront. This is rarely the case for data-driven solutions as they are less predictable than software undertakings and are inherently difficult to estimate and plan. Therefore, this branching strategy is not explained in detail, and an interested reader can refer to Atlassian (2022) for more information.

A branching strategy dictates how source code and project artifacts should be managed in a repository. The next step in a project lifecycle is using the source code to build the resulting solution, deploy it, and have it run in an environment. This enables further collaboration on the project and makes the solution available for other team members, clients, and systems that make part of a company’s IT ecosystem. The upcoming section details commonly found testing stages on a data-intensive project (and what underlying environments are used). These represent a host of activities that take place between development (whose workflow takes place according to a selected branching strategy) and delivery of the solution to the client.

## Self-Check Questions

1. What is the link between a repository and a VCS?

*A repository is a place where artifacts are stored, whereas version control system manages the access to it.*

1. Different branching strategies often use the same type of branches, true or false? Justify your answer.

True, as a branch type represents certain usage of a repository (e.g to fix a bug or implement a feature) which is reoccurring, regardless of the branching strategy employed on a project.

## 3.2 Environments and Stages

Before releasing any artifact to production, which implies it will be consumed by various applications or end-users directly, thorough testing is needed. In the CRoss Industry Standard Process for Data Mining (CRISP-DM) – the most used project lifecycle for data-intensive endeavors (Piatetsky, 2014; Saltz, 2022) – there are two stages that explicitly address testing. During *modeling*, the last step of the process is to assess the model, which means that the person developing it needs to evaluate multiple proposals (e.g., models or variations of its configuration) in terms of the technical measures such as desired effectiveness or accuracy. Once the technical model assessment is done, the project moves on to the next stage – *evaluation*. This is when the client gets involved in the testing process, and the solution is formally assessed against business success criteria. In agile methods, such as Scrum, testing is an ongoing process both for the project team creating the solution and end-users or business stakeholders.

Regardless of the way the testing is approached, the relevant teams need to perform it somewhere. Whether in the cloud or on proprietary physical servers, we speak of **testing environments** that correspond to different versions of the same solution that reside on the underlying infrastructure. Whenever we speak of testing from the point of view of the project lifecycle, **testing stages** correspond to different, consecutive levels of solution verification and validation. Therefore, there is a direct link between who carries out testing activities and how this is done (testing stages) and where these activities take place (testing environments) – please refer to the diagram below for a standard organization of testing.

Relation Between Testing Environments and Types of Tests Executed

Chart, text

Description automatically generated

Rafal Wlodarski (2023)

### Development

As its name suggests, the development environment is used for implementation of a data-intensive solution. It is a workspace that enables the internal project team to make changes without breaking the live environment. Therefore, the customer or business stakeholders should not access it. This is because it always contains the latest version of the code, still under development. It may contain some bugs or simply not represent the final version of the solution that addresses all requirements and business goals.

At this stage, two types of tests are usually carried out: automated unit tests and manual tests that ensure the solution element under development works as indented.

There are no constraints on how a development environment should be managed – it can be created as a shared resource for all the team members working on the solution or each person can have his/her own development environment. The only thing to keep in mind is that whenever a given environment is not used, it should be disposed of for the sake of releasing underlying infrastructure resources (whenever proprietary servers are used) or cost-saving (whenever cloud services are leveraged).

### Quality Assurance (QA)

**Exploratory testing**

A testing procedure that does not follow a predefined structure and focuses on discovery of the solution; the idea is to uncover defects that might not be in scope of other, more formal tests.

Once a part of the system is ready from the developer’s point of view and basic quality checks are performed, the solution moves to the QA environment (also referred to as the testing environment). This is dedicated for the Quality Assurance team as well as any external applications that might consume the model under development (e.g., websites, spreadsheets, dashboards). Therefore, mostly exploratory and integration testing is performed here.

As at this stage the solution interacts with other applications and systems, it needs to be exposed with an API interface that is the same or backward compatible (it can be successfully employed with earlier versions of the system) as the one in production. On the other hand, the underlying infrastructure of a QA environment doesn’t need to closely resemble the production one – oftentimes, fewer machines are used for cost saving purposes.

### Staging

When a model or solution performs well, it can be operationalized and made available to the business stakeholders in a production-like environment. Ideally, the staging environment is a mirror of production, meaning that it is a clone in terms of the underlying infrastructure as well as any dependencies that might exist in production and the various systems that currently consume it (line-of-business or back-end applications). If the solution is deployed in multiple locations in production, the same should apply to the staging environment.

Before going live, all automated and manual functional and nonfunctional acceptance tests need to be run. The objective is to confirm that the deployed model and pipeline address business goals and meet the customer’s needs in a setting that’s as close to real-world usage as possible. Otherwise, some solution properties – think performance or scalability – cannot be efficiently verified, which consequently may lead to many production issues.

### Production

Finally comes the production environment, which is employed by the end users as part of their day-to-day business activity. It exists in every company and, needless to say, is the most crucial environment in an IT landscape. The project team working on a solution will most likely not access it directly (unless there is a critical, operations blocking issue that needs to be addressed), and instead a fresh copy of the production environment is used for troubleshooting and debugging needs.

This section detailed different testing stages**,** whichcorrespond to activities of solution verification and validation and which usually take place one after another as part of a project lifecycle. Therefore, testing stages concern the project delivery process and can build upon different levels and types of tests. In the next section, you’ll learn different ways of distinguishing between testing types and how certain tests constitute a testing stage.

## Self-Check Questions

1. Which environments are always present on a data-intensive projects?

* *Development environment*
* *Quality Assurance*
* *Staging*
* *Production*
* *All of the above*

## 3.3 Testing Strategies

Before we dig into the intricacies of testing data-intensive solutions, let’s demystify the basics of testing. It is defined as a “*process of all project lifecycle activities, concerned with planning, preparation, and evaluation of products or systems, and related artifacts to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose, and to detect defects*” (ISTQB, 2022). The objective of testing technology-driven systems is three-fold:

**Quality attributes**

These describe externally perceived properties of a system as well as its operations. Measuring quality attributes helps to establish how well a system performs its intended functionalities, e.g., with respect to performance, reliability, or security.

1. Validating that the system behavior is in conformance with customer needs.
2. Verifying compliance with requirements specifications.
3. Ensuring that the desired quality attributes are in place.

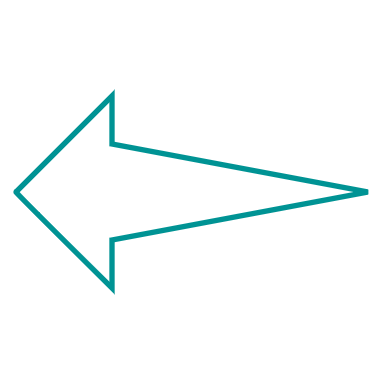
In simple terms, a **testing strategy** is an approach to testing on a given project. Consequently, it implies defining and documenting how to go about meeting the aforementioned objectives.

### Test Levels

One way to approach this is by focusing on **testing levels** – that is the granularity of the system elements that are targeted with tests. Very often, a test pyramid is used to represent the notion of testing levels (see diagram below). As you climb up the pyramid, the cost, effort, and complexity of your tests will rise significantly, whereas their reliability, speed, and isolation will drop.

The lowest level (that is, most granular) is called **unit testing**. A unit is the smallest possible individually executable part of the system – a function, a task, or a procedure. Unit tests verify that a unit works as expected in isolation. A good practice in this area is whenever you encounter a piece of logic with a precise objective, to extract it into a separate function and unit test it.

A Test Pyramid Presenting Typical Test Levels of a Software or Data-Driven Solution



**Increasing cost and effort**

Rafal Wlodarski (2023)

For example, say you wrote a custom Python function (sum\_iterable(object) in the code snippet below) that allows you to sum all values in an iterable object (one that can be looped over, such as a list or a tuple). To check if it works correctly, you could verify that the result it returns is correct using the *assert* statement (test\_sum\_iterable\_list()):

def sum\_iterable(object):

result = 0

for val in object:

result += val

return result

def test\_sum\_iterable\_list():

assert sum\_iterable([1, 2, 3]) == 6

if \_\_name\_\_ == "\_\_main\_\_":

test\_sum\_iterable\_list ()

That’s the basis of unit testing – you write a dedicated function that verifies (via the assert syntax) that a certain condition holds for the code you test – most commonly that boils down to checking the outcome of an operation is equal to a predetermined value.

To effectively employ unit tests on your project, there are a couple more things to learn. Customarily a framework to manage tests is employed so that you do not have to execute them one by one but instead run a whole bunch to see if your code works as expected.

For projects written in Python, [*pytest*](https://docs.pytest.org/en/7.2.x/) is commonly used; to install it on your local machine, simply run the command pip install pytest.

Pytest identifies test scripts by looking for files that have a “test\_” prefix and “.py” extension; within them, functions whose name begin with “text\_” are called (like the test\_sum\_iterable\_list() in the example above). To execute the tests defined on your project, in a terminal window navigate to a folder where your test files are located and run the pytest -v command; you should obtain an output similar to this:

Sample Output of Running Automated Tests With *pytest*

Graphical user interface, text, application

Description automatically generated

Rafal Wlodarski (2023)

By listing all the tests that passed and failed, you can easily locate a problem in your codebase and fix the corresponding function.

Whenever you need to test multiple scenarios of the same operation, rather than writing many similar tests you can *parametrize* the arguments of your function. It’s enough to add a pytest.mark.parametrize decorator to your test as in the following example:

@pytest.mark.parametrize(“test\_input,expected”, [(“3+5”,8), (“2+4”,6), (“6\*9”, 42)])

def test\_eval(test\_input, expected):

assert eval(test\_input) == expected

The decorator defines three different test input–expected output tuples, which will be passed to the test\_eval function.

Another mechanism that will come in handy is called *fixtures*. It allows you to provide a code block that will be called before any of your tests. That is very useful whenever you need to set a context for your tests – a shared environment (for example a database configured with known parameters) or a pre-determined content (such as a dataset or instantiated classes). You can read more on the usage of fixtures in the official documentation of pytest, available on the internet.

Data-driven solutions very often interact with external tools such as databases, or decision support systems. As unit tests verify small executable units in isolation, they should not rely on any external objects. Fortunately, such dependencies can be *mocked*, which means imitating an operation or object that your code depends on. A prime example is having a call to a database that is expected to exist in a production setting but for obvious reasons mustn’t be used for testing purposes. Mocking such a connectivity means telling Python to return a certain value instead of making the actual call to the production database. Such an approach makes it possible to write and run unit tests without requiring a connection to external systems.

Unit tests are very useful to detect bugs in your functions; however, it is nearly impossible to cover the integrity of your system with them. Therefore, other tests are needed as well.

Next, there are **integration tests**. They ensure that different stages of a pipeline interact correctly or that services, libraries ,and tools used by your solution work well together. The idea of integration tests is to detect any issues whenever different system elements (either internal or external) interface with each other. Integration testing implies combining components with each other to verify their behavior as a subsystem and check the integrity of intermediate results.

There is no single recipe for integration tests – their structure and elements under test depend on the context of your solution. Imagine you are working on a system that leverages machine learning and includes three pipelines:

1. A feature generation pipeline (used both at training and serving time), which processes raw data to produce a feature vector.
2. A training pipeline, used to train a machine learning model.
3. An inference pipeline, which uses the trained model to return predictions from new observations.

A simplified diagram of the system looks like the following:

A Machine Learning Solution With Integration Points Marked

Graphical user interface, diagram

Description automatically generated

Rafal Wlodarski (2023), based on (Ploomber.io, 2022).

Integration tests verify if inputs and outputs flow seamlessly through the system’s components until the final result is generated. Therefore, in the described scenario the points that are susceptible to being tested are: 1) any intermediary result that’s produced by one component and consumed by another one, 2) any intra-pipeline interactions.

With regards to 1) we need to ensure that the input expectations of the next processing task are the same as the output of the previous task, otherwise the flow could be broken. One way to address that is by defining a function that verifies the anticipated format and calling it on the resulting intermediary file. Common checks include column names, numerical columns within specific ranges, categorical values within a pre-defined set of values, and no empty values.

With regards to 2) we need to make sure that the serving pipeline correctly integrates with the model file generated by the training pipeline. The underlying test should check if the inference pipeline produces the same features that the model was trained on. That would help detect a human error such as updating only one of the pipelines with newly added features and not the other one.

Finally, at the very top of the pyramid we find ourselves with **acceptance tests**. These are formal tests executed to verify if a system satisfies its business requirements. This implies that the entire application needs to be up and running, hence all the solution components/modules need to be combined. The idea is to verify proper functioning of the integrity of your application, by replicating target user behaviors and interactions with the system.

They are sometimes referred to as end-to-end tests, although this is just one implementation of acceptance testing. They allow testing the flow of a solution from start to finish (from the data source to the final result) to see if it is performing as designed.

An acceptance test suite acts as a quality gate to decide whether a product is ready for release or not.

### Test Types

So far you learned about testing levels, which denote granularity of the solution elements under test (unit, integration, system). Usually, the operations provided by the system need to be performed in a certain way – maintaining a desired level of performance or respecting given security measures. For the system to be considered successful, it needs to support a set of quality attributes. Therefore, a desired characteristic of the solution needs to be tested. We say that such an attribute becomes the main concern of testing or the test objective. This is another *testing strategy* – focusing on a desired quality attribute or concern of the solution and executing corresponding **test types** that allow it to be verified.

We speak of *functional tests* whenever user/business requirements are verified and *non-functional tests* when presence of a quality attribute is checked. Another common type of test is the *regression test*, which verifies that after some changes have been introduced in the system, it still behaves the same way as prior to the modification.

A literature review by Feleder et al. identifies *data quality* and its properties – correctness in particular – as one of the main test concerns in data-intensive projects. Whenever low-quality input is provided to a data pipeline, most likely the resulting outcome will be of equally low value. While a definition of clean data depends on the context of your project, some common checks should be implemented as a series of tests. Recurring data issues that should be addressed include null values, duplicates, evolving raw data properties, and incorporating invalid observations in the analysis. Such verifications are good candidates for integration tests.

Another crucial aspect of data-driven solutions is the underlying *model quality*. Defining its success criteria and the assessment itself should be achieved in the modeling phase of a project lifecycle structured according to CRISP-DM.   
Training a model is not a single-stage procedure; therefore, defining a mechanism for determining whether tests were successful or not might be very tricky. In practice, one way to approach the problem is to define test criteria relative to previous results. Before deploying a new model, it should be benchmarked with the current solution in production to ensure that its performance doesn’t deteriorate.

Let’s take as an example a solution that provides predictions based on regression analysis. We will use mean absolute error (MAE) as the model evaluation metric and compute it across the validation set as well as a sub-population of interest. Then, new features are added to the model and the model is trained. The resulting metrics are as follows:

|  |  |  |
| --- | --- | --- |
| **Validation set** | **Benchmark MAE** | **Candidate model MAE** |
| Complete | 1.4 | 1.2 |
| Sub-population 1 | 0.5 | 0.4 |
| Sub-population 2 | 2.0 | 2.1 |

Rafal Wlodarski (2023), based on (Ploomber.io, 2022)

It seems that the new model has improved the results of MAE, which are reduced for the complete data set as well as the first sub-population. Nevertheless, the metric increased for the second population subset, so you might wonder if the model is indeed better. The assessment is not conclusive based on a single set of reference values. Therefore, the models are trained multiple times with the same parameters to generate a distribution of results. The outcome is the following:

|  |  |  |
| --- | --- | --- |
| **Validation set** | **Benchmark MAE** | **Candidate model MAE** |
| Complete | 1.4, 1.2, 1.3 | 1.2, 1.1, 1.2 |
| Sub-population 1 | 0.5, 0.5, 0.6 | 0.4, 0.4, 0.3 |
| Sub-population 2 | 2.0, 2.3, 2.4 | 2.1, 2.0, 2 |

Rafal Wlodarski (2023), based on (Ploomber.io, 2022)

This empirical distribution can be used to verify if the MAE values of the candidate model are within the observed range. It could be done programmatically by following the procedure indicated in the following pseudocode:

def test\_model\_quality():

#load upper and lower limits of reference values

min\_ref, max\_ref = load\_ref\_values()  
 candidate\_model\_val = load\_current\_values()

assert min\_ref <= candidate\_model\_val <= max\_ref

From such a test, we can monitor whether the performance of the model drops or increases. We should look out for sudden changes as compared to the reference values in any direction. Big drops in metric values might stem from incorrect processing of data (hence some unintended modifications were made somewhere along the pipeline) or accidentally dropping rows/columns. Large gains should be analyzed cautiously to see if the root of the change is not information leakage rather than an actual improvement such as adding new features or more training data.

Other common concerns of model quality that should be addressed with tests include verifying that abnormal data points – e.g., known product faults still get classified as “faulty” – or simple outliers are treated as expected.

Typical Test Levels and Types on a Data-Driven Project

Radar chart

Description automatically generated

Rafal Wlodarski (2023)

To summarize, there are various testing levels (covered in the previous section) and testing types, which each deal with specific concerns of a data-driven solution. These dimensions are orthogonal (as in the figure above) but complement each other. Some system tests can verify functionalities of the solution, while others might seek to ensure that the solution can process a data set of given size within a desired timeframe (hence verify the performance of the system). There are no strict guidelines on how testing levels and types should intertwine, and defining a sufficient set of tests should emerge from your project context. Both functional requirements and business drivers need to be addressed with tests so that a quality solution is delivered to your client.

The guiding principle is to increase testing levels as the development of the project progresses. In its early stages, focus should be put on unit tests that cover small pieces of your solution. Integration tests should be put in place whenever different project artifacts need to cooperate and start forming a pipeline or when external systems are involved. Finally come acceptance tests, which reflect business expectations of the solution and mirror its target usage. Keep in mind that the complexity and associated effort to implement testing increases through these stages, in turn requiring different technical/business skills and timeframes for completion.

One way to minimize the latter is to leverage available packages and libraries. That particularly applies for complex data transformations or advanced algorithms. Third-party solutions are tested before release to the public (just like you ensure testing of your project before delivering it to the client), which reduces work for you.

### Self-Check Questions

1. Should a complete testing strategy be based on different levels *and* types of tests? Why?

*Yes, as both address different aspects of testing – namely granularity and concerns – so are complementary rather than mutually exclusive.*

Summary

In this chapter, you’ve learned the basis of collaboration on data-driven undertakings. A source code repository is the central archive for all project artifacts. A version control system (VCS, like Git) makes it possible to store the history of changes to a repository and enables efficient management of stored artifacts.

A branching strategy defines an approach as to how the team should perform operations on a project repository. Different branches can be created to represent specific needs and are managed according to a workflow, such as *centralized* or *feature-branch*. Each of these defines a predetermined set of steps to follow to integrate changes into a repository. Once a certain part of the development is considered done, the corresponding source code and artifacts can be used to create a version of the solution that ultimately will be released to the client and used in production.

However, before that happens, the artifact goes through multiple environments that are employed as part of specific testing stages. The idea is to make a running solution available to different teams that verify that it works as planned from many perspectives. These standpoints are encompassed by a testing strategy that answers the non-trivial question, how will the solution be tested on this project?

There are two major ways to approach testing. The first is according to testing levels, which represent varying granularities of the system elements that are targeted with tests (i.e., unit/integration/acceptance). On the other hand, there is the testing objective – a desired quality attribute or concern of the solution – that a test aims to check. On data-intensive projects, you’ll often encounter functional (verifying user/business requirements), model quality, and performance tests.

# Unit 4 – Continuous Integration/Continuous Delivery

**Study Goals**

On completion of this unit, you will be able to …

… name and describe possible stages of a CI/CD pipeline.

… identify which automation steps are relevant to organize a data-driven project with Continuous Integration/Continuous Delivery.

… implement a CI/CD pipeline that encompasses multiple stages and employs containerization.

# 4. Continuous Integration/Continuous Delivery

## Introduction

Data-intensive systems are complex, and a simple change to a single file can have unintended side effects on the overall system. Therefore, when a team of developers work on related systems, the risk of breaking others’ code increases significantly. Have you already worked in a small team during your studies? Remember some of the obstacles encountered? As you worked together as a team, most probably you have unintentionally impacted your peers’ code and introduced a bug. Possibly in more than just one place.

What’s more, as data-driven solutions involve a large number of files, including code, tests, and external dependencies and libraries, coordinating artifact updates can be a daunting and costly task. It is often referred to as “integration hell.”

To address such issues, back in the 1990s Microsoft introduced nightly builds followed by a set of automated tests. These were the underpinnings of *continuous integration* (CI). So, what has CI become today? It simply means a practice that automates a part of the project development lifecycle.

And how about *continuous delivery* (CD)? This deals with the same concern but pushes the automation a step further, by preparing a release-ready solution. You’ve probably heard both terms many times as they are very hot in the industry right now. With the growing popularity of CI and CD, there are high chances you will work with them during your career, so read this chapter carefully and get a head start on the subject!

## 4.1 Concepts of a CI/CD Pipeline

Automation of a project development lifecycle is achieved by putting in place what we call a CI/CD pipeline. This is essentially a runnable specification of steps performed to build a new version of a data-intensive solution. It means that instead of doing it manually, the process is written programmatically and can be re-run whenever needed.

The constituent elements of a pipeline are represented by the acronyms that precede its name: CI and CD. Let’s demystify them one by one.

CI is unambiguous and is an abbreviated form of **continuous integration**. “integration” stands for the process of pulling pieces together, making sure different project artifacts bundle correctly and the result is a functioning deliverable. In practice this translates into combining the right versions of libraries, specific toolkits, and data sources, and running automated tests that verify that the outcome is as expected.

CD part means either **continuous delivery** or **continuous deployment**, depending on how advanced an organization is in terms of the automation process. The next step after integration is releasing a project. This activity requires incorporating all necessary project files (source code, libraries, configuration files, etc.) and preparing a runnable solution that the end-users or other systems can consume. This means a specific format is expected that makes it possible to install the solution on a designated environment. A simplistic example of the release process is having a Python package uploaded to the Python Package Index so that other data scientists can use it.

Finally, continuous deployment stands for the process of automatically replacing an existing solution in each environment (usually *production*), with a newly generated artifact (as the outcome of the continuous integration and delivery steps). Continuous deployment on data-intensive projects can be challenging as the output might not be a single model but rather multiple candidate models that need to be evaluated against each other. To make matters more complex, if a model currently deployed in production turns out to be superior to the new candidate models (e.g., in terms of its predictive power), then no deployment should take place. Continuous deployment is out of the scope of this coursebook, but awareness of it is useful in order to avoid any terminology-related confusion.

To sum up, continuous delivery enables a new version of the solution to be released anytime, but the deployment still requires manual steps (like transferring artifacts from a repository to a production server). Continuous deployment builds on the automated release process and automatically (without human intervention) takes care of the deployment of project artifacts to the target environment.

Having clarified the integral elements of a CI/CD pipeline, let’s see what stages can constitute them.

### Building Blocks of a CI/CD Pipeline

A typical CI/CD pipeline encompasses following stages that are automatically executed in a sequential order: trigger, code check out, compilation, automated testing, code analysis, and notification (see figure below). You might not necessarily encounter all of them in every CI/CD pipeline – each project is unique and employs different processes and associated technical practices.

Stages of a Generic CI/CD Pipeline

A picture containing text

Description automatically generated

Rafal Wlodarski (2023)

Let’s see what each of these steps looks like in practice.

#### Step 1: The trigger

The call to initiate the CI/CD process can come from a remote script or another pipeline, but most frequently it’s the source control management system, such as GitHub that automatically triggers the pipeline upon a push to the repository.

#### Step 2: Code checkout

Then, the CI server retrieves the code corresponding to the commit in question.

**Step 3: Complication**

Afterward, the actual build stage starts – compiling the solution. The source code and its dependencies are combined to build a runnable instance of the model or corresponding artifacts. As a result of this stage, an artifact that can be run and tested needs to be produced.

This step represents the integration part of a CI/CD pipeline.

#### Step 4: Automated testing

The next key element of any CI/CD pipeline is testing. This stage acts as a safety net that prevents bug and regressions from reaching the end-users.

Multiple QA methods can be performed to that end: unitary, integration, and acceptance tests, which verify that the system behaves as expected at different levels.

Additionally, for data-intensive solutions it is paramount to verify model quality every time changes to its underlying code or related artifacts are made. Some examples include optimization of data transformations, increasing the training set size, or incorporating additional features. Having the tests executed regularly allows you to pinpoint gains (or degradations) in performance from a particular action taken. If this is not done, you might have to train an older version of a model and compare the results. This can be a time-consuming and daunting task if tests have not been run in a long time and many precedent model versions require analysis.

#### Step 5: Code coverage and quality stage

The next CI pipeline stage is optional and is commonly seen on software projects. Nevertheless, for data-driven solutions that rely heavily on code and stress the importance of its quality, a variety of code analysis mechanisms can be invoked:

* Code coverage, which reports on the not-tested sections of the code.
* Code style formatter, to align coding style with industry-standards.
* Code quality verification, which employs metrics such as code size or complexity level to identify places where source code can be improved.

#### Step 6: Release artifact(s)

Depending on the kind of project, the application is then packaged and uploaded to a repository (like GitHub or a container registry). There it can further processed and deployed to a staging or production environment.

This stage corresponds to the delivery part of a CI/CD pipeline.

#### Step 7: Notify success or failure

An important step in a CI/CD pipeline is confirming that the changes were integrated correctly; therefore, the system should produce a clear “pass” or “fail” notification.

Upon encountering any issues, logs need to be provided so that the developer can troubleshoot the problem, fix it, and commit it to the repository to relaunch the whole process.

### Complementary Technical Concepts

Before moving to the next section, where a basic CI/CD pipeline will be put in place, the reader needs to understand some additional concepts and terms that will be used throughout the example. If you’re already familiar with the notion of an automation server (and available solutions of this on the market) and containerization using Docker, and you have put in place a model as a service, you can jump directly to section 4.2.

#### Automation server

A CI/CD pipeline is nothing more than a set of automated processes around compiling, testing, and packaging a data-driven solution. To execute them, a machine with computing resources is needed to do the heavy lifting. That’s exactly what a (build) automation server is for, sometimes referred to as a continuous integration server.

Every automation server exposes programmable objects or applications that allow pipeline steps to be executed. A person in charge of setting up a pipeline needs to indicate what commands he/she wants to run, and the automation server adopts these. Depending on the automation solution used, the encoding of the commands takes different forms; nevertheless, most of the tools used a declarative syntax.

The idea is to indicate what is needed by declaring different sections and parameters, as well as commands to execute. This is an approach very similar to infrastructure as code (IaC) with the tools like Terraform and ARM templates.

Given the fast-growing adoption of CI/CD, there are a wide variety of tools for automation of the delivery process. The one most employed is Jenkins (Leszko, 2019), while CircleCi, Bitbucket Pipelines, GitHub Actions, and GitLabs CI remain popular options. You can refer to Atlassian (2022) for an overview of different automation servers.

**Containerization**

As you already know, an automation server provides computing resources for a CI/CD pipeline to build, test, and deliver an artifact of a data-driven solution. This means that whichever dependencies the application requires on a local machine, they will also be needed on the server. Consequently, all libraries and tools have to be installed every time a CI/CD pipeline is run. Fortunately, there is a far more convenient and increasingly popular solution – using a container that has all the dependencies present.

*Containerization* allows an application to be packaged into an image that can be run on any operating system. Hence, it responds to the problem of how to prepare a stable build environment that will behave the same way regardless of the place (think different automation server solutions) and time of execution.

To ensure this, each container constitutes an entire runtime environment. Therefore, it typically includes:

* the application itself
* any dependencies it might need to be built or tested
* configuration files that are needed to run the app.

**Docker** is an open-source project designed to help with application deployment using containers. It is a market-dominating technology provider for containerization (Kane 2018, Krief 2022). Before using a Docker image as part of your CI/CD pipeline, you need to understand two more notions.

First, Docker distinguishes between an image and a container. An image is the fundamental building block in the Docker platform. It contains all files required to run your application as well as the steps that define how to run it. An image does not store data concerning its state, so you store it in a Docker registry, name it, version it, or save it as a file.

While you could just keep your image locally, there are some common configurations used for different classes of applications, so there’s an interest for them to be public and reused by other developers. Imagine you create a machine learning model – you’ll probably need Python, pip, pandas, numpy and others present in your development environment. There is a place where you could find a Docker image that provides you all these standard dependencies – it’s called a Docker registry. Docker Hub is the Docker company’s public repository of Docker images.

The next thing you need to understand is the distinction between an image and a container. An image is static, and when it is started, it becomes a container. This implies that it’s possible to initiate multiple containers using the same image.

Given that containers are stateful (as opposed to images), you can modify them, e.g., by adding new dependencies as the need arises.

Finally, a point to keep in mind is that to use Docker, it needs to be present on your automation server. Given the popularity of the tool, it is the default set up if you employ cloud-based solutions; otherwise, you need to install it locally on your server.

#### Model as a service

A **service** represents an **abstraction** for describing executing programs. The concept of model as a service is to provide an easily employed *interface* that other systems and end-users can consume and receive results in a format that is easy to parse. With this approach, the implementation details of a model (the how – which programming language, commands and data structures, libraries or packages were used to achieve a result) are hidden. Only entry points that a developer defines, and which respect commonly used conventions, are visible to the external world.

**Interface**

A shared boundary between different systems, components, devices, or humans that is used to exchange information. Software interfaces hide all information and resources and allow selective access via well-defined entry points.

Moreover, all services share common characteristics:

* They are accessible over the network and can be invoked using standard protocols.
* They perform only a single function.
* They are relatively quick to provide a result.

Services differ in terms of how they exchange information – the communication protocol and the data format used. Within the scope of this coursebook, only **RESTful** services are discussed, which employ HTTP and return results in a JSON structure.

Working with REST implies that the structure of a service’s interface needs to be defined (and documented) so that others know how to use it. Standard elements include:

* HTTP URL (the endpoint used to call a service)
* A list of parameters and their types
* JSON response body that specifies the names and value types that constitute the result returned by the service.

All of the aforementioned points describe the end-result – what a model should look like so that it can be easily used by others. You can already tell that it differs greatly from running a model in a notebook such as a Jupyter or Rstudio.

In order to make a model accessible programmatically, a simple approach is to persist it as an API (application programming interface) and run it on a server.

For that purpose, the Flask micro-framework is a common choice among the data science community. It will be used in the CI/CD pipeline example, so if you’re not familiar with it, please refer to one of beginner-level tutorials: basic (Dholakia, 2020) or more comprehensive (Krebs, 2022).

## Self-Check Questions

1. What can CD stand for in CI/CD?

*Continuous Delivery or Continuous Deployment.*

1. Continuous Delivery is the last stage of a fully automated project development lifecycle.

True/False

## 4.2 Building a CI/CD Pipeline for Data-Intensive Systems

In this section, a three-stage CI/CD pipeline for a data-driven solution will be set up. The sample project is a model written in Python and used for classification of a bank’s customers. Based on their characteristics –age, occupation, balance, etc. – it needs to identify customers that are likely to accept a loan offer. Once the training process is finished, the model is exposed as a service on a Flask server.

The CI/CD pipeline will consist of the following three stages:

1. Unit testing
2. Release artifact
3. System testing.

Pipeline steps can be encoded in different ways, depending on the tool stack used. For the sake of this example, GitLabs will be used as the underlying automation server. It is a cloud-based tool that is integrated with any GitLabs code repository. It is a convenient solution as it allows the development, testing, and delivery of a project to be managed in one place. It also provides a Web IDE (a browser-enabled editor) for the pipeline definition file.

### Working With GitLabs CI

A pipeline definition is written into a .gitlab-ci.yml file, which is stored in the root directory of a repository. It is a YAML file where you configure specific instructions for your GitLab CI/CD pipeline. You can create the file locally and push it to the repository or use the Web IDE and navigate to the “CI/CD -> Editor” menu as per the screenshot below.

CI/CD Pipeline Configuration Screen in GitLabs CI

Graphical user interface, application, Teams

Description automatically generated

GitLabs CI offers templates for many commonly used technology stacks, and Python is one of them. You can select it or create an empty file as all the necessary elements of the YAML file are discussed below.

As described in section 4.1 (Building blocks of a CI/CD pipeline), a pipeline’s operations are regrouped into stages. Hence the first thing to specify in the pipeline definition file is the **stages** element along with its constituents. The next stage is initiated only when the previous one is complete, according to the order of their definition after the “stages” keyword.

A GitLabs pipeline starts automatically when a commit is made to your GitLabs repository, so there is no need to specify the trigger and code check out stages. Therefore, the CI/CD pipeline of the example project contains only three stages, specified in the .gitlab-ci.yml file as per the snippet below.

# List of stages for jobs, and their order of execution

stages:

- unit-test

- release-api

- system-test

Keep in mind that indentation is important, and any global keyword (refer to the [documentation here](https://docs.gitlab.com/ee/ci/yaml/)) that configures pipeline behavior needs to be followed by a new line and indentation. Comments are preceded by the hash (#) sign.

Stages allow certain types of actions in a pipeline to be grouped, but it is the “Job” element that performs the actual work. Jobs specify operations that are performed by the automation server (GitLab Runners are used for that purpose).

*Jobs* are the way you define automated tasks that your CI/CD platform executes. Each job needs to:

* be part of a pre-defined stage (as indicated by the “stage” keyword)
* contain at least the “*script*” clause, which indicates a command to be executed.

### Setting up Jobs in GitLabs CI

#### Unit tests job

The first job in our CI/CD pipeline – unit testing – would therefore look like this:

unit-test-job:

stage: unit-test

script:

- apt-get install -y python3-dev python3-pip

- pip3 install Flask gunicorn pytest pytest-cov

- pytest tests

The script section contains three commands:

1. apt-get install, to ensure project prerequisites – Python and pip – are present on the underlying, Linux-based system
2. pip3 install, to ensure project dependencies are present
3. pytest tests to run the pytest framework

One very important thing to keep in mind is that all the operations performed as part of a job are not persisted and are not accessible outside of its scope. This means that although you’ve prepared a working environment for a Python application in steps 1 and 2, these resources are only available within the “unit-test” job, and they need to be repeated every time the same job is relaunched. Later on, you’ll see how this clearly sub-optimal approach can be addressed using containerization.

There are a couple of points to take care of so that the unit test stage of our CI/CD pipeline can be executed smoothly. As tests will be run on a model exposed as a service on a Flask server, the FLASK\_APP environment variable needs to be specified so that the framework knows how to load the application.

The most convenient and transparent way to set up environment variables for a GitLabs CI/CD pipeline is through a dedicated settings page. You need to navigate to “Settings -> CI/CD”, scroll to the “Variables” section, and select “Add variable” as in the screenshot below. It is enough to fill in the key (“FLASK\_APP”) and value fields (corresponding to the name of your app as declared in the code, e.g., app = Flask(“app”)). To learn more about environment variables, please refer to [GitLabs documentation](https://docs.gitlab.com/ee/ci/variables/#add-a-cicd-variable-to-a-project).

Environment Variables Configuration Screen in GitLabs CI

Graphical user interface, text, application

Description automatically generated

Rafal Wlodarski (2023)

The Pytest framework requires use of its syntax to declare tests as well as a certain organization of files (e.g., having all tests in a dedicated folder at the root level of a project). Please refer to Gathuku (2022) for basic tutorial on Pytest, to Hillard (2022) for a more comprehensive one, and finally to Zolkepli (2018) to see a sample Pytest project and how it’s reflected in a GitLabs repository.

The next job in your pipeline will take care of the continuous delivery or CD part of the process and releases a runnable artifact. You could choose to serialize the trained model using a framework (e.g., Pickle for Python, see Jurney, 2017 for an example), store the resulting file in a repository, and use it for further deployment. In the example, a Dockerized version of the service will be generated instead. The reasoning is two-fold – practical use of containers is one of the learning objectives of this script, and secondly, the image will be employed again in a later stage of the sample CI/CD pipeline.

#### Generating a Docker image

In order to build a Docker image, a file containing the set-up of the target solution is needed. At the root directory of your repository, create a file with the name “Dockerfile”.

A Dockerfile is a text document that contains the instructions to assemble a Docker image. As a reminder, an image includes everything needed to run an application – the code or binary, runtime, dependencies, and any other file system objects required. Hence, to prepare such an image, you need to write commands that allow such dependencies to be prepared.

Please note that all the commands marked in bold represent Docker-specific instructions, whose full reference can be found in the official documentation on Docker’s website.

The first mandatory step is to tell Docker what base image will be used for the target solution. This is achieved by adding a line with the **FROM** keyword. For the purpose of a model written in Python, “debian:11” which is a free, universal operating system, will be used.

Then, all project dependencies need to be installed using the **RUN** keyword. This tells Docker that the command that follows is to be executed on the base image. Therefore, the command provided needs to be compatible with the operating system of the image used. In the example, the Linux command “apt-get” is used for installation of Python3 and pip (its package installer). Afterward, pip is used to install dependencies of the sample project that is built with Python.

The snippet below shows how to define the base image that will be pulled from a container registry (debian:11) and that is followed by multiple installation steps as outlined above. These steps prepare a foundation for the solution execution.

# syntax=docker/dockerfile:1

FROM debian:11

RUN apt-get update && apt-get install -y \

python3 \

python3-pip

RUN pip3 install Flask

RUN pip3 install gunicorn

RUN pip3 install requests

#more solution-specific dependencies might be needed

Next, the solution source code (and any other required files) needs to be added into the image. The command **WORKDIR** sets the location of a working directory – the starting point of any operations that will follow. Therefore, you might want to pick the root folder of your project (“app” in the example). The **COPY** command duplicates files into the Docker image. Its first parameter is the file to be copied (“.” stands for all files), and the second one is the target folder inside the image.

The snippet below tells Docker to use a directory called “app” as the working directory of the image. Then all the files (“.” parameter of COPY) that exist wherever you created the Docker file are copied to the target working directory (“app” parameter).

WORKDIR /app

COPY . /app

Finally, all there is left is to tell Docker what command should be used to run the solution when the image is executed inside a container. This is achieved with the CMD command. For a Flask server to be started properly and be accessible over the network outside of the container, it needs to bound to the non-routable meta-address 0.0.0.0, with any port (number 3000 is used throughout the example).

The snippet below sets the working directory of the image to “app/web” (this is where the runnable file is placed), opens port 3000 to the container that will be run from the image (EXPOSE 3000 command), and finally provides the command to start the container (CMD instruction and everything that follows).

WORKDIR /app/web

EXPOSE 3000

CMD ["gunicorn", "-b", "0.0.0.0:3000", "app:app"]

It’s important to keep in mind two points concerning the CMD instruction. First, there can only be one CMD, and if that’s not the case, Docker will use only the last one that’s present in the Docker file. Second, the parameters that are passed to CMD correspond to the command that’s used to run your solution. Therefore, they are solution-specific, and the possible errors or issues might relate to it or the set up rather than the Docker platform.

As a consequence, it is strongly advised that you create and test your Docker image locally. Do this gradually to make the troubleshooting more effective – add instructions one at a time, and check if the output is as expected or some issues occur. Please refer to the official documentation for Docker Desktop installation as well as the following materials: simple image building tutorial, overview of Docker for image creation and MongoDB tutorial, resolving problems with Dockerfile.

With the Docker file finalized and present in the root folder of the repository, let’s get back to setting up the next stage of our CI/CD pipeline – releasing an artifact.

#### Release artifact job (continuous delivery)

As usual you need to start by adding a new element to the stages section and define a job that will be executed. A high-level objective of the release artifact job is to build a Docker image containing the model under development and push it to a repository (save the image to a registry so that it can be accessed at any time).

Using Docker commands requires Docker to be present in your CI/CD pipeline. To enable it, add the “image” keyword and the name of the image of your choice. Now, it will be the Docker container that executes the job’s script and not the automation server directly.

**Artifact**

An artifact is any tangible item produced as part of your solution. Depending on the need and the purpose it serves, it can take any form – from an executable file encompassing the whole solution, to a Docker image that’s used for testing, to a document that helps the team implement the solution.

To build a Docker image, you also need to start a Docker daemon, which runs in the background and executes the commands that are specified in the script section. In case of GitLabs, one way to start a Docker daemon – the simplest one – is by using Docker-in-Docker as a service. This translates into adding the “services” keyword and declaring the docker image we want to use, where the “dind” part stand for Docker-in-Docker.

The services keyword allows you to start additional Docker containers and to link them to your image. One typical use-case of such a mechanism is when you need to access a database, an API, or some other service as part of a job.

It may all sound quite complicated; therefore, to get a better understanding of these concepts (although this is not necessary as the commands provided are sufficient to run similar data-driven solutions), please familiarize yourself with GitLabs CI (2022a), and Despa (2021).

For now, let’s recap the current configuration of the .gitlab-ci.yml file and move on with the next steps of the release artifact stage.

release:

stage: release-api

image: docker:20.10.16

services:

- docker:20.10.16-dind

Before you can build and push images using GitLab CI/CD, you must authenticate with a *Container Registry*, GitLab’s repository of Docker images.

This is achieved by executing the Docker login command. Since the plan is to use the command from a GitLab CI/CD pipeline, there is no need to know or generate any credentials. You can simply reference the CI\_REGISTRY\_USER, CI\_REGISTRY\_PASSWORD, and CI\_REGISTRY environment variables.

The generated image could be pushed to any other Docker registry, e.g., the one provided by Docker – Docker Hub – or other providers. Then, the environment variables would need to be created to provide actual credentials used to authenticate with the external system.

The next, key step is building the image by executing the “docker build” command. Two parameters are provided:

1. “-t”, which provides a name and tag of the image. You can keep the pipeline configuration-free by using the predefined GitLab CI variable CI\_REGISTRY\_IMAGE.
2. The build context directory, in the example denoted as “.” – current directory.

The build context directory is the directory on the host machine where docker will get the files to build the image. Docker build will use the Dockerfile located in that directory. What’s more, all files from that directory will be visible to docker build – so the COPY command used within your docker file refers to that directory.

When you tell Docker to build an image by executing the docker build command, Docker reads the instructions prepared in the Dockerfile, executes them, and creates an image as a result. This is then pushed to the GitLab Container Registry and visible on the corresponding screen.

Containers Registry in GitLabs CI

Graphical user interface, application, email, Teams

Description automatically generated

Rafal Wlodarski (2023)

See the below snippet of the complete specification of the release artifact stage.

release:

stage: release-api

image: docker:20.10.16

services:

- docker:20.10.16-dind

script:

- docker login -u $CI\_REGISTRY\_USER -p $CI\_REGISTRY\_PASSWORD $CI\_REGISTRY

- docker build -t $CI\_REGISTRY\_IMAGE .

- docker push $CI\_REGISTRY\_IMAGE

#### System test job

Now that the Docker image containing your data-driven solution has been built and stored in a Docker registry, system testing can be performed.

Before the corresponding configuration is discussed, let’s pause for a second and consider alternative approaches to automated system testing. It examines the entire, integrated solution (as opposed to its underling unitary functions or components) so needs to account for communication between different components of an application or external systems. How do you go about this issue when writing unit tests?

By using doubles such as stubs and mocks. This approach can also be employed in system testing. A *mock* replaces an object the system depends on with a test-specific object that verifies that the system component is using it correctly. A *stub* replaces an object the system depends on with a test-specific object that provides test data to the system component.

Such an approach is costly, as mocks need to be created and maintained as the system evolves. What’s more, it brings limited value to the customer as the actual flow of data through the system is not tested.

Can you think of another approach to system testing? It is testing against a deployed environment. This implies connecting to real test instances of a back-end, other models, or third-party systems your solution communicates with. However, this implies preparing and managing dependencies there, changing behavior depending on the machine, and ensuring data consistency, among many other headaches.

What would be yet another approach that, as you should remember by now, takes care of all application dependencies and is easily reproducible and maintainable?

It’s testing against a Dockerized version of a component or service. A Docker image contains a model as a service (with all its dependencies), which means it can be easily and readily used, e.g., as part of a CI/CD pipeline.

In large scale projects, system testing usually requires running a dedicated test suite that checks the behavior of the system. For the sake of simplicity, in the example, acceptance testing will be performed by calling the web service endpoint (which corresponds to, e.g., a dashboard or website using the model to generate predictions).

The first steps are very similar to the previous job: A new stage needs to be added, its name defined, and the underlying image declared as a docker-in-docker service.

In the script section, Docker also needs to login to a Docker registry so that the image can be pulled – this is done in the same way as in the previous example. Pulling the image is implicitly done – retrieving the created image from GitLabs Containers Registry – whenever the ”docker run” command is executed. This is when the image is instantiated and becomes a running container. In turn, this container runs your application.

The “docker run” command parameters are

* “–d”, which stands for “detached mode” and means running the container in the background
* “–p”, which stands for “port” and specifies a container’s port(s) to the host, and
* the name of the image you want to run, which is stored in the GitLabs variable $CI\_REGISTRY\_IMAGE.

Now that your solution is running in a container, it can be tested. As it is exposed as a RESTful service, communication with the model is executed by accessing exposed URLs and providing expected parameters. In the case of the bank customers classifier, their features need to be provided, e.g., age, job, and account balance.

**Curl** is a command-line tool to transfer data to or from a server, and that’s what will be used to call the web service exposed in the container. A prior installation step in the underlying docker image (“- apk add curl” command) is necessary. Curl can call HTTP POST methods (which the bank classifier service accepts) and provide required parameters with -XPOST and -F parameters. The outcome of the query can then be verified (here the possible answer is “YES” or “NO”) and the successful outcome of the stage determined.

See the snippet below for the complete specification of the system test stage.

run-curl-system-test:

stage: system-test

image: docker:20.10.16

services:

- docker:20.10.16-dind

script:

- docker login -u $CI\_REGISTRY\_USER -p $CI\_REGISTRY\_PASSWORD $CI\_REGISTRY

- docker run --name api -d -p 3000:3000 $CI\_REGISTRY\_IMAGE

- apk add curl

- curl -XPOST http://docker:3000/classifyClient’ \

-F ‘Age=25’ \

-F ‘Martial=single’ \

-F ‘Balance=456.23’ \

-F ‘Housing=no’ \

-F ‘Education=primary’ \

| serviceAnswer

* test serviceAnswer == (“YES” || “NO”)

## Summary

In this unit you’ve learned that a CI/CD pipeline allows automation of parts of a project development lifecycle, with the last possible step being deployment of a data-driven solution. Depending on the project context, different stages can constitute a pipeline; however, it is customary to perform several types of testing as well as preparing an artifact that can be deployed to an environment of choice. This process begins whenever changes are pushed to the project repository and ends upon the successful execution of all pipeline stages. Each CI/CD pipeline stage needs to be programmatically encoded and define commands to be executed without human intervention.

A set of technical concepts and tools underpin an effective delivery process with CI/CD:

* an automation server that provides resources for it
* containerization, which makes it more resources-efficient and easily reproductible
* model as a service, which enables easier automated testing.

When working with CI/CD pipelines, it is important to be meticulous about dependencies management and rely on Docker images to avoid re-executing unnecessary set up steps.

Finally, as a CI/CD pipeline provides a lot of flexibility in terms of its organization (granularity, presence, and order of stages) it is important to consider different solutions that are adapted to a given project context (usage of containerization or not, approach to system testing, etc.).

Self-Check Questions

1. A CI/CD pipeline can release an artifact ready for deployment to which environments?

* *Development environment*
* *Quality Assurance*
* *Staging*
* *Production*
* *All of the above*

# Unit 5 – Container Communication and Networking

**Study Goals**

On completion of this unit, you will be able to …

… discuss how APIs and containerization enable agility of the delivery process.

… compare different networking drivers for Docker and their use cases.

… choose a networking set up adapted for the context of your project and apply it locally or in a cloud environment.

… explain what container orchestration is and when it is needed.

## Introduction

There is an increasing recognition that data-driven projects need to move from small-scale experimentations to large-scale solutions that can be readily deployed and scaled. To enable this transition, models should be treated more like software rather than science. That implies versioning their iterations in a central repository, ability to execute automated tests before releasing any artifact to the client, and finally packaging the solution in a form that makes it easy to deploy and run on any machine. All of the above is possible if models are treated as services and delivered as containerized APIs (application programming interfaces).

*Model as a service* implies developing a model and creating an interface for it that other systems and end-users can consume programmatically and receive results in a format that is easy to parse. Containerization encapsulation of that API, any dependences it requires, and configuration files needed to run it in an image. Such an artifact can then be easily distributed and deployed at scale in any environment. This is in a sharp contrast to the standard project delivery life cycle.

A data-driven solution moves through multiple environments, which might have differing configurations in terms of libraries, binaries, etc. This means that a model could behave differently in development and staging environments. A solution that worked while testing may not work in production. This in turn usually leads to a time-consuming and costly investigation of why one instance of the same application behaves differently to another one.

Containerization addresses the portability issue by isolating an application and its dependencies so that it can run on any platform, irrespective of the operating system or the underlying infrastructure. Consequently, it solves the problem of how to reliably run and scale a data-driven solution across differing computing environments, thus enabling an organization’s true agility.

## 5.1 Containers and APIs

To understand the notion of containerization, you can think of a physical shipping container. It is a box that allows you to store some goods and distribute them across the globe in a streamlined fashion using many transportation modes: trucks, trains, ships. The operators of these machines can move the container around without knowing what is inside as the content itself is isolated and contains everything needed.

A software container is similar as it contains something self-sufficient – an image encompassing an application and all of its dependencies and configuration files – that can be run everywhere in the world. The component that fills the cranes and transportation mode role is called Docker. It is a software platform that allows applications to be reliably developed, shipped, and run as containers.

**Virtualization**

Is a process that allows the emulation of physical computing resources – an entire machine down to the operating system and hardware layers – under the form of a virtual machine. As a result, it is possible to have multiple virtual machines run on a single computer.

**Containers** provide an **image-based deployment model** to share an application together with its dependencies, but without the operating system. The solution interfaces directly with the host operating system, so there is no additional layer of a guest operating system. This results in better performance and no waste of resources as compared to virtualization (for more material on the differences between the two, please refer to Buchanan (2022)).

Docker Platform on a Host Machine

Graphical user interface, application

Description automatically generated

Rafal Wlodarski (2023)

The diagram above depicts Docker’s architecture mapped to the elements of a host machine. You can see that the Docker platform resides directly on a machine/server (once installed there) and communicates with the underlying resources via a component called the Container Engine. This is an application that creates and manages Docker objects, such as images and containers, represented by the top two layers. It uses the Linux kernel, and features of the kernel such as namespaces, to segregate processes so they can run independently. Don’t worry, you don’t need to understand the technicalities behind Docker’s functioning to be able to use this technology efficiently – that is the whole beauty of it.

### Application Programming Interface

The acronym API stands for **application programming interface**. In the context of this definition, the word application refers to any technology-driven solution with a distinct function. Interface can be thought of as a contract of service between two applications. This contract defines how the two communicate with each other using requests and responses.

To better understand what an API is, let’s use a real-world analogy again, this time of a restaurant. In a typical restaurant, there are three essential actors that make the business run: a waiter, a chef, and a client. Think of the waiter as the API (the messenger). He/she is the person in charge of delivering specific information to the right end. A waiter receives information from a client and delivers it to the chef. Then, once the kitchen finishes preparing an order, once again the waiter will be the one bringing the dish to the table. Their job is just to transfer an order to the chef and once it’s done, bring the result (a cooked dish) to the customer.

To sum up the restaurant analogy, an API has a similar role to a waiter: it provides a means of communication between a client (a system or software) and the kitchen (another system serving the requests) that prepares the orders.

In more technical terms, an API encompasses the following:

* an abstraction of a solution implementation (details of how an executing program works are hidden)
* specification of data types used by the API
* semantics to model the exchange of information.

These elements constitute a solution representation that allows developers and other systems to implement an API consumer.

An API exposes a set of data and functions that other computer programs access via a well-defined interface. The API consumers usually perform an API call that involves exchanging information over a network or the internet. This is whenREST (which stands for representational state transfer) comes in. It is an architectural style that, in the simplest terms, defines standard ways of communication over the network using HTTP protocol. An API conforming to REST is called a REST API and consists of interlinked resources that are accessible via HTTP end points.

As depicted in the diagram below, an API is the face of a data-driven solution, directly listening and responding to client requests that traverse a network.

REST API Request/Response Handling and a Restaurant Analogy

Graphical user interface

Description automatically generated

Rafal Wlodarski (2023)

Now that you understand the terms containerization and API, you must be wondering what the link between the two is?

Both represent a method to “package” an application in a way that makes the underlying compute resources easily accessible and more portable. An API wraps a data-driven solution into a form that makes it programmatically accessible to anyone who knows its interface (URL endpoint and parameters) and the format of the expected result. Containerization on the other hand bundles the runtime ecosystem of the solution itself – everything that is needed for it to execute and become operational as an API. This means that the dependencies that are required for your solution to work – e.g., visualization or data manipulation libraries, frameworks, programming languages – are packaged along with the model and no longer need to be installed or maintained on the server. Use of APIs and containers in conjunction increases the overall agility of the project delivery process and contribute to an organization’s competitiveness by reducing time-to-market. This is why it is important for you to be familiar with these concepts at a high level when managing data-driven solutions.

## Self-Check Questions

1. What does API acronym stand for?

*Application Programming Interface.*

1. What is the difference between a docker image and a docker container?

*Docker image is a static snapshot of a development environment while a container is a is a running instance of a Docker image.*

## 5.2 Container Orchestration and Networking

Although containerization is an enabling technology, using it in large-scale systems – think management of hundreds of containers – can become a challenge when done manually. Hence an additional layer of automation is introduced in such cases: container orchestration. In the production context, it ensures that processes are resilient and efficient, in turn making the operational complexity of dealing with a large number of containers manageable.

Like infrastructure as code and CI/CD, container orchestration uses a declarative syntax to automate containers’ lifecycle and associated activities: scaling, load balancing, and networking. To achieve this, a configuration file is written (in YAML or JSON depending on the tool), which defines a desired configuration state.

A container orchestration configuration file typically includes:

* definition of container images that make up a system along with their location
* provisioning information for the containers: storage and other resources
* definition of network connectivity between containers
* versioning information.

The orchestration tool runs the file and schedules deployment of the containers (and optionally their clusters) on a host machine to achieve the declared state. Once the containers are put in place, the orchestration tool manages their lifecycle based on the container definition file (very often a Dockerfile). The lifecycle-related activities include the following:

* management of scalability, load balancing, and allocation of compute and storage resources among the containers
* distribution of containers among the host machines to ensure availability and desired level of performance
* collection of metrics and storage of log data to enable monitoring of the health and performance of the system.

There are dedicated orchestration tools on the market, which provide a framework for management of containers and their lifecycle at scale. Docker includes Docker Swarm, which is the platform’s own container orchestration tool. However, the tool that dominates the market is called Kubernetes (IBM 2022). Most leading public cloud providers – Amazon Web Services (AWS), Google Cloud Platform, IBM Cloud, and Microsoft Azure – offer managed Kubernetes services. The tool, in the broadest terms, helps companies to fully implement and rely on a container-based infrastructure in production. As it is quite advanced and typically operated by DevOps teams, you are unlikely to use it to manage small-scale projects. Nevertheless, if you would like to learn the tool, you can refer to some high-quality external materials: Burns et al. (2019), or for a more succinct overview with AWS, Krief (2022).

### Containers Networking

In today’s connected world, it is hard to imagine any device that is not part of the internet network. The same applies to data-intensive solutions – they do not operate in isolation but rather need to communicate with other applications, systems, and databases, which usually happens over the network. In this section, you’ll learn the basics of management of container communication in a network.

Containers are separate network entities within the operating system. This means that every running Docker container has a unique IP address assigned to it on the host machine. You can check it from a shell session within the container (exec command). It is enough to run the command hostname -I as per the following code snippet.

$ docker exec -it *container\_name* /bin/bash

Having an IP address enables communication between different containers via a simple TCP/IP link. However, this is just the most straightforward scenario of container networking, and much more sophisticated needs may arise in an IT ecosystem – defining virtual networks for a group of containers or connecting a single container to multiple networks. Such advanced set-ups require special mechanisms to allow connections to and from containers.

There are many types of network configurations that can be implemented with containers; however, it is enough to be familiar with a handful of them to cover the most common use cases. Below you’ll find a description of some widely used container networks along with tutorials on how to set them up.

#### None

In this scenario, containers operate in a fully isolated network, which implies that no communication with the external world can take place. While it might not be evident, this simple set-up can be useful when testing containers or running containers as part of batch processing to produce a result during one of the steps.

#### Bridge

The bridge network is the default network set-up in Docker that allows external network connectivity. For this purpose, it uses IP tables to provide network address translation (NAT) between the container and the host network. Furthermore, it enforces network isolation between the two so that multiple applications that use the same container port within a single host can be run.

Naturally, all containers running within the single host can communicate using the container IP addresses. Nevertheless, communication with containers running on a different host is not possible with the bridge network, unlike in other set-ups. Similarly, incoming communication from the external network is not allowed.

Network Setup of “Bridge” type

**Diagram

Description automatically generated**  
Rafal Wlodarski (2023)

In the tutorial two containers will be run on the same Docker host, and by executing some tests, their communication will be demonstrated in the Bridge network. For all the tutorials to come, Docker needs to be installed and running on the machine where the set-up takes place.

In step one, using Docker desktop in a new terminal window, list current networks with the command “docker network ls”. In a clean environment, you should see three networks (with differing IDs): *bridge*, which is the default network, as well as *host* and *none* (which are described in this section).

**Text

Description automatically generated**

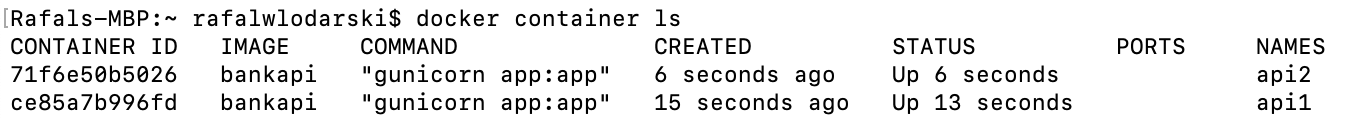
In step two, two containers (of any image you wish) are created with the following parameters: “*-dit*”, which employs the detached (background) and interactive mode, with a TTY (this allows you to see input and output of the container), “*—name*”, which specifies the name of a container you will run.

$ docker run -dit --name api1 bankapi

$ docker run -dit --name api2 bankapi

There are two things to keep in mind at this stage: First, as the containers are started in the detached mode, their ID will be displayed after you type the docker run command instead of a connection to the container taking place. Second, given the lack of a “--network” flag, the containers use the default bridge network.

In step three, run the “docker container ls” command to check if the containers were correctly started.



In step four, to verify that the containers are connected to the bridge network, the inspect command can be run (which returns detailed information on any Docker objects): “docker network inspect bridge”.

Text

Description automatically generated

Under the “containers” key, you should see two entries that correspond to the containers you’ve started (verify the “name” field) along with their IPs. Another interesting piece of information is the IP address of the gateway between the bridge network and the Docker host machine (“IPAM/Config/Gateway” node: 127.12.0.1).

In step five, as the containers are still running in a detached mode, use the docker attach command to connect to your container: “docker attach api1”. As a result, the prompt sign changes to #, which means you are the root user within the container.

**Gateway**

A hardware or software component that sits between two networks. It converts information, data, or protocols to enable communication between the networks.

In step six, to verify that it is possible to emit outgoing signals from your container to the external network, you can ping any public web site. In the example, google.com will be used with a flag limiting the ping attempts activated: “ping -c 2 google.com”.

The results shows that a request was successfully delivered to the target address.

Text, letter

Description automatically generated

Now, let’s verify that communication between the containers is possible by using the ping command again. This time around, it’s the second container’s IP that is the target address (you can retrieve it from step 4): “ping -c 2 172.17.0.3”.  
Similarly, a successful operation is expected given that in the bridge network, containers residing on the same host machine can communicate freely.

Text, letter

Description automatically generated

In step seven, to leave the attached mode, use the CTRL +p, CTRL + q combination. You can then verify that communication in the opposite direction is possible (from container 2 to container 1) by repeating steps 5, 6, and 7 and by substituting the IP address and name of the containers.

#### Host

A Host set-up employs the network namespace of the host machine, which means that NAT is not needed. As a consequence, it enables fast network communication between containers and makes it possible to accept incoming traffic from an external network. The containers have access to the host machine’s physical interfaces. This implies that whenever a client residing in an external network wants to communicate directly with a container, it can do so by using the host machine’s ports. The downside of this set-up is that there is no network isolation between containers, and port-conflict issues may arise and pose a security threat.

Network set up of “Host” type  
Diagram

Description automatically generated

Rafal Wlodarski (2023)

In this tutorial, a container will be started and bound directly to the Docker’s host network on port number 80. As in the host set-up, there is no network isolation: the underlying process of the container appears as if it was running directly on the Docker host machine (rather than in a container).

There are two prerequisites for the tutorial to work:

1. The port number 80 needs to be available on the Docker host.
2. The host driver is supported only on Linux hosts and not Docker Desktop for Mac or Windows.

For the sake of the example, nginx image will be used, but you can go ahead and employ any image you wish.

1. Start the containers with the following parameters: “*--rm*”, which removes the container once it is stopped, “*-d*”, which activates the detached (background) mode, and “--network”, which specifies which network should be used. The full command is as follows: “docker run --rm -d --network host --name my\_nginx nginx”.
2. Navigate to the address <http://localhost:80/> on any browser. A webpage saying “It works!” should be displayed.

This test corresponds to accessing an application running in a container, via the Docker host’s port. It proves that the container can be reached from the external network, which is not possible in the bridge network set up.

**Overlay**

An overlay network uses a networking tunnel to enable communication between containers running on different host machines. Overlay creates a dedicated network layer that sits on top of the host-specific networks. This is particularly useful when working on a cluster configuration using orchestrator tools such as Kubernetes or Docker Swarm.

Network Setup of “Overlay” Type

Diagram

Description automatically generated  
Rafal Wlodarski (2023)

In the tutorial, two containers running on separate Docker hosts will communicate using an overlay network. Docker Swarm will be used to orchestrate a cluster of two docker hosts.

**Cluster**

A set of machines that work together and are viewed as a single system. With Docker, it means multiple docker hosts joined using swarm mode.

As two independent machines are required, Docker desktop cannot be used, and viable alternatives include setting up local virtual machines or cloud-based hosts. In the example, Amazon Web Services will be employed. Regardless of the approach chosen, each host machine needs to have the following ports open:

* TCP port 2377
* TCP and UDP port 7946
* UDP port 4789

In order to set up a virtual machine on AWS, you can follow the tutorial Amazon (2022), steps 1–3. As by default, AWS resources do not have Docker installed, you need to make sure it is present on the virtual machine before proceeding. To achieve this, after step 2f, in the advanced details section, populate the user data field with the following script, which will install and run Docker. All of the stages apart from Amazon (2022) step 2 need to be executed twice to set up two virtual machines.

Installation of Docker on an AWS virtual machine

Graphical user interface, text, application, email

Description automatically generated

Rafal Wlodarski (2023)

Following the tutorial and additional steps outlined above, you should be able to connect to each VM using a terminal session (via the AWS Systems Manager/Fleet Manager menu). To verify that Docker was properly installed, run the “docker -v” command:

Graphical user interface, text, application

Description automatically generated

1. On host1 machine set up a Docker swarm using the following command: “sudo docker swarm init”. This will initialize a group of virtual machines that are running the Docker application and are configured to join in a cluster.
2. Please keep in mind that the sudo part is added because the default user on AWS can’t access the docker engine; hence commands are run as the superuser.

As a result of the swarm initialization, a message indicating how to join the cluster as a worker should be displayed. Copy it and move on to the next step.

Text, website

Description automatically generated

**Tip**: if you would like to know the token but do not have access to the swarm init output, you can always run the following command: “docker swarm join-token worker”. Alternatively, if you are looking for the manager token, change the last item in the command to “manager”.

1. In the terminal session of the host2 machine, run the docker swarm command (retrieved in the previous step) preceded by “sudo”.
2. On host1, set up a new overlay network with the following parameters: “--driver=overlay”, which specifies the driver type, and “--attachable”, which allows standalone containers to join the network. The complete command is as follows: “sudo docker network create --driver=overlay --attachable test-net”.

You can run “docker network ls” to verify that the network was successfully created (look for a name that corresponds to the title specified after “--attachable”):

A picture containing text

Description automatically generated

1. On host1, execute the command “sudo docker run -it --name alpine1 --network test-net alpine” to start a container that connects to the “test-net” overlay network in interactive mode (“-it” parameter).
2. Similarly, on host2 start a container in the same way but in the detached mode using the command “sudo docker run -dit --name alpine2 --network test-net alpine”.
3. On host1 within the interactive terminal, use the ping command to verify if container running on host2 is reachable: “ping -c 2 alpine2”. A successful outcome should look like the following:

Text

Description automatically generated

1. Once you’re done with the tests, make sure to stop the containers, remove the network, and stop the virtual machines to avoid incurring unnecessary costs.

In the last scenario that you’ve set up, two standalone containers run on two cloud-based virtual machines and communicate over a network of overlay type.

Familiarity with the four network drivers discussed is sufficient for most of the use cases you’ll come across on small-scale projects. Should you be interested in more advanced techniques, you can refer to the official documentation (Docker, 2023) on Docker networking, which also contains hands-on tutorials.

## Self-Check Questions

1. Bridge is the default network set up in Docker: True or False?

*True*

1. What is the name of family of tools used to manage a lifecycle of a big cluster of Docker containers? What are its prominent examples?

*Orchestration tools; Docker Swarm, Kubernetes.*

### Summary

In today’s technology market, agility and shipping solutions at speed are essential ingredients of success. Containerization and APIs cater to this as they help to streamline the delivery process.

An API is a set of definitions and protocols used for integrating applications in a programmatic way. APIs make your product or service easily accessible over the network without the consumer having to know how they’re implemented.

Wrapping a model in an API usually requires the presence of multiple dependencies (think libraries, frameworks, binaries) on the machine where it will be run. Containerization solves the portability issue as it allows the creation of a self-sufficient image of the runtime ecosystem that can be easily run anywhere. This allows the containerized solution to be moved between different environments or cloud platforms with almost no effort.

While in small numbers, containers are easy enough to deploy and manage manually, nowadays many organizations rely on a significant and rapidly growing number of containerized applications. Efficiently and reliably handling containers at scale therefore calls for automation. Orchestration tools like Kubernetes and Docker swarm help DevOps teams to manage the lifecycle of containers and integrate them into CI/CD workflows. Along with application programming interfaces (APIs), containerized services are the foundation for cloud-native applications.

# Unit 6 – Tools for Managing Data Projects

**Study Goals**

On completion of this unit, you will be able to …

... name and describe types of tools that support management of data-driven projects.

... classify tools among different project development lifecycle activities.

… examine different approaches to selecting a project tool stack and choose the one adapted to your context.

## Introduction

Modern approaches to project delivery promise increased agility, higher solution quality, and shorter time to market. They also rely heavily on automation, and automation is made possible by software and tools. Therefore, to enable the proclaimed benefits, agile project management and DevOps require putting in place technology tools that serve as means to implement associated processes and best practices.

The resulting toolchain should be able to address multiple stages of a project lifecycle, tackle DevOps fundamentals including continuous integration (CI) and continuous delivery (CD), and allow teams to collaborate all around. On the operations side, tools should help with effective monitoring and incident management. As a consequence, a DevOps toolchain is oftentimes a collection of tools that operate as an integrated unit to design, build, test, deploy, and operate data-intensive solutions, as well as providing continuous feedback to the teams involved.

Given the wide array of activities underpinning a project lifecycle and the possible ways to automate or enhance it, the market has responded with a plethora of both specialized and more generic solutions. This chapter will give you an overview of the different types of tools that are involved in technology-driven solutions delivery and general guidelines on how to choose the right tool stack for your project context.

## 6.1 Overview of Tools for Managing Data Projects

DevOps is considered as an evolution or natural next stage of agile project management. Both represent a cultural shift for a company – a new way of working as well as the technology that helps to implement it.

When adopting new practices and tools, one must consider many factors so that the change initiative does not backfire. There are technical aspects to take into consideration — how does the new piece of software integrate with the existing IT ecosystem? How will it communicate with the next component in the new toolchain? The best solution out there is useless unless it can be integrated effectively.

There are also human factors — is the tool easy to learn, well documented, and has an active support community, so that the team can adapt easily? Often, organizations experiment with different combinations of tools so that they can find just the right end-to-end solution for their context. Of course, a trial-and-error approach can be costly both in terms of time and financial resources; therefore, it is better to take an informed decision in the first place.

### Choosing a Tool Stack for your Project

There are two primary approaches to selecting a DevOps toolchain: an all-in-one solution or an open toolchain. Naturally, each has some trade-offs associated with it, but an important thing to understand is that they both represent overarching decisions that will structure how the development landscape will look in the long run.

#### All-in-one solutions

An all-in-one solution represents a software suite that covers multiple stages of a project lifecycle. This means that a wide variety of activities can be accessed through the same interface and by design can interact seamlessly. Most commonly, two types of companies offer such software. First, version source control providers like GitHub, BitBucket, and GitLab, as all the project development phases rely on a source code repository, so providing additional capabilities allows them to have a bigger share of your business. Second, cloud providers, as they propose a wide variety of services in general and also want to have as much of your business as possible.

The advantages of using an all-in-one solution include relatively straightforward set-up and a lower learning curve in general (after all, there is only one tool to manage), easy integration between different stages of a CI/CD pipeline, and a large community of users. With regards to the downsides, such solutions may not easily integrate with other, third-party tools and as they try to address many concerns and needs at the same time, might be more or less performant in specific areas (say strong in repository management but not so much in issue tracking).

**Legacy system**

A technology, system or application that’s outdated but still in use. In industrial IT ecosystems, there are often some parts that cannot be replaced, and the new elements need to interact/integrate with these.

As a consequence, all-in-one solutions are a good fit for teams that are new to agile/DevOps or are simply looking to start a project rapidly. On the other hand, mature teams that already have a preferred set of devices might have a hard time switching to a completely new tool stack. Dealing with legacy technologies and tools can also inhibit the use of all-in-one solutions.

#### Open toolchain

The opposite, “open” approach to a DevOps toolchain is having dedicated tools for different parts of a CI/CD pipeline or solution development activity. This allows the tools most adapted to a particular project context to be selected, or the team’s preferences to be catered to. As adoption of agile approaches and DevOps is constantly on the rise, more and more specialized tools are available on the market making it more possible to choose a best-of-breed solution that addresses all your needs.

Integration is an essential part of an open toolchain – having multiple tools that don’t integrate means operational nuisances for team members. Their day-to-day activities translate into switching between different screens, logging in to multiple places, and oftentimes manually synchronizing information between the tools, which naturally leads to degraded efficiency and lower satisfaction.

The pros of working with dedicated tools are that they usually offer rich functionality and a high degree of customization and in principle are better in providing their specialized capabilities when compared to all-in-one solutions. However, when many such tools need to be put in place, high start-up effort is needed (researching/trying out different tools, installation, and set-up), and one should be wary of potential integration headaches (proofs of concept are always recommended to make sure that all the tools can be glued together to constitute a full CI/CD pipeline).

Consequently, an open toolchain is a good fit for experienced teams that have well defined needs and might want to tailor their way of working (and as a result the corresponding tools) by teams and/or by project. Having a hand in the choice of specific, customizable solutions allows companies to incorporate their existing technologies into a wider DevOps toolchain.

### Tools for Specific Activities in a Project Development Lifecycle

Regardless of the project management method that’s followed, a data-driven solution goes through a set of stages that represent different activities needed to turn a concept into software running in production. Using the open toolchain approach means that a dedicated tool might be chosen for a set of specific tasks in the development lifecycle. This section gives an overview of popular tools that are available on the market.

#### Plan and track

In order to efficiently manage a team project, everyone involved needs to know what tasks lie ahead and how they interrelate so that dependencies can be anticipated. So-called issue tracking tools allow you to do just that – break work down into smaller, manageable chunks, push them through a workflow (e.g., to do, in progress, in test, done) and monitor progress toward the project goals. As the name suggests, they also enable the continuous gathering of user feedback, organizing it into bugs or improvements, and prioritizing resulting actions for the team. Finally, storing user requirements is also supported to a lesser or greater extent.

Prominent tools in this category include **Jira** (the market leader), **Confluence**, **Trello**. A point to keep in mind is that every major version control tool (**GitHub, Bitbucket, GitLab**) offers at least basic issue tracking functionality so should be considered for smaller projects or whenever a simplified workflow is sufficient.

Popular Issue Tracking Software



Rafal Wlodarski (2023)

#### Build

The actual development of a data-driven solution may require multiple purpose-specific tools. First, there is an IDE (integrated development environment) where the actual coding takes place and which is usually linked to the technology used – this category is out of scope of the coursebook. Then, there is source control or version control software, which allows changes to the code to be stored in a central repository. This is also the enabling technology for collaborative project efforts as every team member can have his/her own version of the code base and share it with others.

**Integrated Development Environment**

A software tool whose main purpose is to enable efficient code development. Some underlying functionalities include editing, building, testing, and packaging of solutions.

When it comes to source control tools, there are three companies that dominate the market: GitHub, BitBucket, and GitLab. To see a detailed comparison between these tools and their usage contexts, please refer to Alexsoft (2022a).

Popular Source Control Software

Logo

Description automatically generated

Rafal Wlodarski (2023)

#### Testing

Automated testing not only verifies that a solution works as expected but also underpins any continuous integration/continuous delivery pipeline, making it an essential part of the DevOps tool stack. There are a wide variety of tools in this area that serve different purposes. First there are frameworks that work with different programming languages and allow executable tests to be prepared (e.g., **Pytest** and **Robot** for Python). Other tools specialize in a specific type of testing – e.g., **Snyk** and **Mend** for security testing or **Selenium** for UI tests. Finally, dedicated tools have been devised for test management (planning testing activities, overseeing their execution, and reporting on their status) with the likes of **Zephyr Squad**, **Xray**, or **TestFLO** for Jira.

#### Continuous integration/continuous delivery

CI/CD tools take care of building a solution (as in integrating code changes from multiple contributors), running automated tests on it, and generating a releasable artifact. Different stages of the delivery process are structured in a customizable pipeline that sequentially executes its underlying steps.

There are dedicated tools that automate the end-to-end release management lifecycle (**Jenkins** being the most popular solution; **CircleCI** is another example); however, many companies rely on cloud providers (AWS CodePipeline, Azure Pipelines, or Google’s Cloud Build) or their version control tool. The latter usually comes with a module/toolset that seamlessly integrates with the repository itself (GitHub Actions, Bitbucket Pipelines, GitLab CI/CD) and hence is a go-to option for smaller projects.

For an overview of dedicated tools for CI/CD as well as their usage contexts, please refer to Alexsoft (2022b).

Popular Continuous Integration (CI) servers

Circle

Description automatically generated with medium confidence

Rafal Wlodarski (2023)

#### Deployment and resources provisioning

Automating a solution’s deployment means programmatically shipping a runnable artifact to a given environment and can usually be achieved with the same tooling used for the preceding stages of a CI/CD a pipeline. Nevertheless, two supporting technologies take the automation process a step further and pave the way for DevOps adoption – containerization and infrastructure as code (IaC).

Containerization means packaging a solution (or an environment) along with its dependencies into a standalone image that will execute the same way no matter the configuration of underlying resources. Whenever management of a significant number of containerized applications is needed, orchestration tools are introduced. They automate associated tasks of scalability, load balancing, and allocation of compute and storage resources among the containers as well as taking care of logging and metrics collection. There are two dominant forces in the containerization market: **Docker** for creation of the containers themselves and **Kubernetes** for their orchestration.

Infrastructure as code tools allow the IT infrastructure needed to run data-driven solutions to be provisioned programmatically – that means databases, compute resources, load balancers, etc. As a result, the process is more consistent and reproducible across many environments. Popular IaC tools on the market include: **Terraform, Puppet, Progress Chef, Ansible.** For a more fine-grained distinction between such tools (and more alternatives), please refer to Brikman (2019).

Popular Infrastructure as Code tools A picture containing graphical user interface

Description automatically generated

Rafal Wlodarski (2023)

Let’s take a closer look at two tools that are leaders in their respective markets and which represent the two aforementioned approaches to DevOps tool stack selection: an all-in-one solution (GitHub) and an open one (Jenkins).

## Self-Check Questions

1. Can you employ both all-in-one solution and dedicated tools to constitute a DevOps tool stack? Why yes/no?

*Yes – an all-in-one solution does not need to cover all project development activities (even if the tool makes it possible) and can be supplemented with specialized tools.*

1. Issue management software can be used all along a project development lifecycle – from requirements elicitation to implementation to maintenance.

True/False

1. Which of the following tools overlap in their offered capabilities?

* *GitHub*
* *GitLab*
* *BitBucket*
* *All of the above*

These represent source control tools that also provide CI/CD capabilities.

## 6.2 GitHub

A code repository is fundamental to any collaborative, data-driven endeavor as it is as an archive for all the information needed to deal with the revisions and history of a project. However, this is not enough – a tool that manages and tracks different versions of a solution is needed, and that’s exactly what GitHub does. This version control system (VCS), purchased by Microsoft back in 2018, is the most commonly used tool of its kind and represents more than 83% of the market share (Slintel, 2023).

No wonder the company expanded its activity to respond to the growing popularity of DevOps and now also serves as an automation platform for CI/CD. The corresponding toolset is called **GitHub Actions** and is a direct competitor to specialized tools like Jenkins and TravisCI.

GitHub Actions is a prime example of an **all-in-one solution** that provides a set of related components (*a repository, VCS, an automation server*) that can be conveniently put together to execute customizable automation workflows in an understandable and predictable way. The key word here is conveniently, as GitHub Actions provides a seamless (accessible via the same user interface and carried out in the same tool under the hood) and flexible experience that allows you to create an end-to-end CI/CD pipeline that corresponds to your project’s needs.

It is important to note that the “CD” stands for *continuous delivery*, and for a solution to be made available to users, an extra step is needed – deployment. A common use case here would be to use one of the cloud providers for storage and computing resources needed, and in a fully automated workflow to employ an IaC tool like Terraform to provision those resources.

One of advantages of using a widely popular tool like GitHub is its community of users. This not only means a large number of people that know how to use the tool and are willing to disseminate that knowledge, but also a dedicated platform – the Actions Marketplace – where reusable solutions to workflow automation are shared. As a consequence, rapid prototyping is enabled as you can leverage common CI/CD pipeline set-ups instead of starting from scratch.

To get started with the tool, you can refer to an introductory tutorial (Douglas, 2022) and deep dive into any topic as needed in Laster (2023). A potentially interesting read for you, which shows additional capabilities and integration possibilities of the tool, is an article by Rizel (2022).

## Self-Check Questions

1. What is the difference between GitHub and GitHub Actions?

*GitHub is a repository and as such serves only for storage, management and collaboration on source code. GitHub Actions is an integrated tool that builds on GitHub and enables Continuous Integration/Delivery.*

## 6.3 Jenkins

Jenkins is an open-source automation server and is one of the most popular tools for implementing CI/CD processes (Laster 2018; Leszko 2022). Similarly to GitHub, it has active community-based support as well as offering a significant number of plugins (over a thousand) that allow you to customize the tool for the context of your project. It is language-agnostic as it supports most programming languages and frameworks (via the plug-ins mechanism).

Another key feature of Jenkins is that it allows you to treat *pipelines as code*. This means that you can programmatically declare your CI/CD process (in YAML/XML files), and the tool will execute the underlying steps automatically (a similar mechanism to the one you’ve seen in units 2 & 3). Configuration of the tool itself can be achieved the same way. As a consequence, everything you need to set up and run your CI/CD pipeline can be created, updated, and version-controlled along with the rest of the solution source code.

As discussed earlier, specialized tools that can constitute an open DevOps tool stack oftentimes pose integration issues. Jenkins addresses them head-on by being a flexible and portable tool at many levels:

* It supports all major version control systems (VCS) – GitHub, BitBucket, and GitLab – and integrates with many other external systems and tools via plug-ins.
* It is cross-platform, which means it can be run on any operating system (Windows, Linux, and macOS). It is distributed in a variety of versions – Web Application Archive (WAR) files, Docker images, Kubernetes operators, and OS-specific binaries, which makes it easy to install and leverage any type of computing resources.
* It can be deployed both on premise or in the cloud through a simple configuration executed in a user interface or in the command line.
* It is available as both a free, open-source tool and an enterprise, paid edition (offered by CloudBees, [www.cloudbees.com/jenkins](http://www.cloudbees.com/jenkins)).

The unique set of features and capabilities discussed has allowed Jenkins to become the most used software of its kind (Laster 2018; Leszko 2022). Nevertheless, it is important to underline that Jenkins must be hosted on a server to provide those features. Be it on a proprietary server or in the cloud, this requires some initial set-up (refer to [this tutorial](https://www.jenkins.io/doc/tutorials/tutorial-for-installing-jenkins-on-AWS/) for a taste) as well as maintenance efforts. This often becomes the main adoption barrier for teams that do not have profiles with infrastructure skills among them. Having said this, Jenkins is considered easy to learn and operate (Leszko 2022), so it is not at a disadvantage when compared to other available tools of the same kind (CircleCI, Atlassian Bamboo, JetBrains TeamCity).

To get started with Jenkins, you can refer to the comprehensive official documentation (Jenkins, 2023), a hands on-book (Leszko, 2022) that uses the tool in a wider CI/CD context, or advanced reading if the administration angle is relevant for you (Laster, 2018).

## Self-Check Questions

1. Can one use Jenkins jointly with cloud-solutions e.g. provided by Azure?

*Yes, Jenkins can be deployed as part of cloud-provided resources.*

## Summary

Agile and DevOps ways of working aim to speed up the development life cycle, make it highly collaborative and automated to the largest extent possible. This, however, is not possible without a range of supporting tools. Choosing the right tool stack is therefore critical to the long-term success of a project and should be based on multiple factors: team needs, integration capabilities, ease of learning and use, pricing, and work-time efficiency.

There are two major approaches to deciding on the project tool stack: go for an all-in-one solution or choose specialized tools for different activities that underpin your development process. The first one is associated with a smaller learning curve and set-up cost, which makes it a good fit for immature teams (in terms of DevOps knowledge and experience) and small projects. An open tool stack grants a large degree of freedom and customization and hence responds to particularities of workflow/team/project needs and technological constraints (like legacy tools). Nevertheless, the key point to keep in mind is interoperability of the tools chosen and the integration effort required.