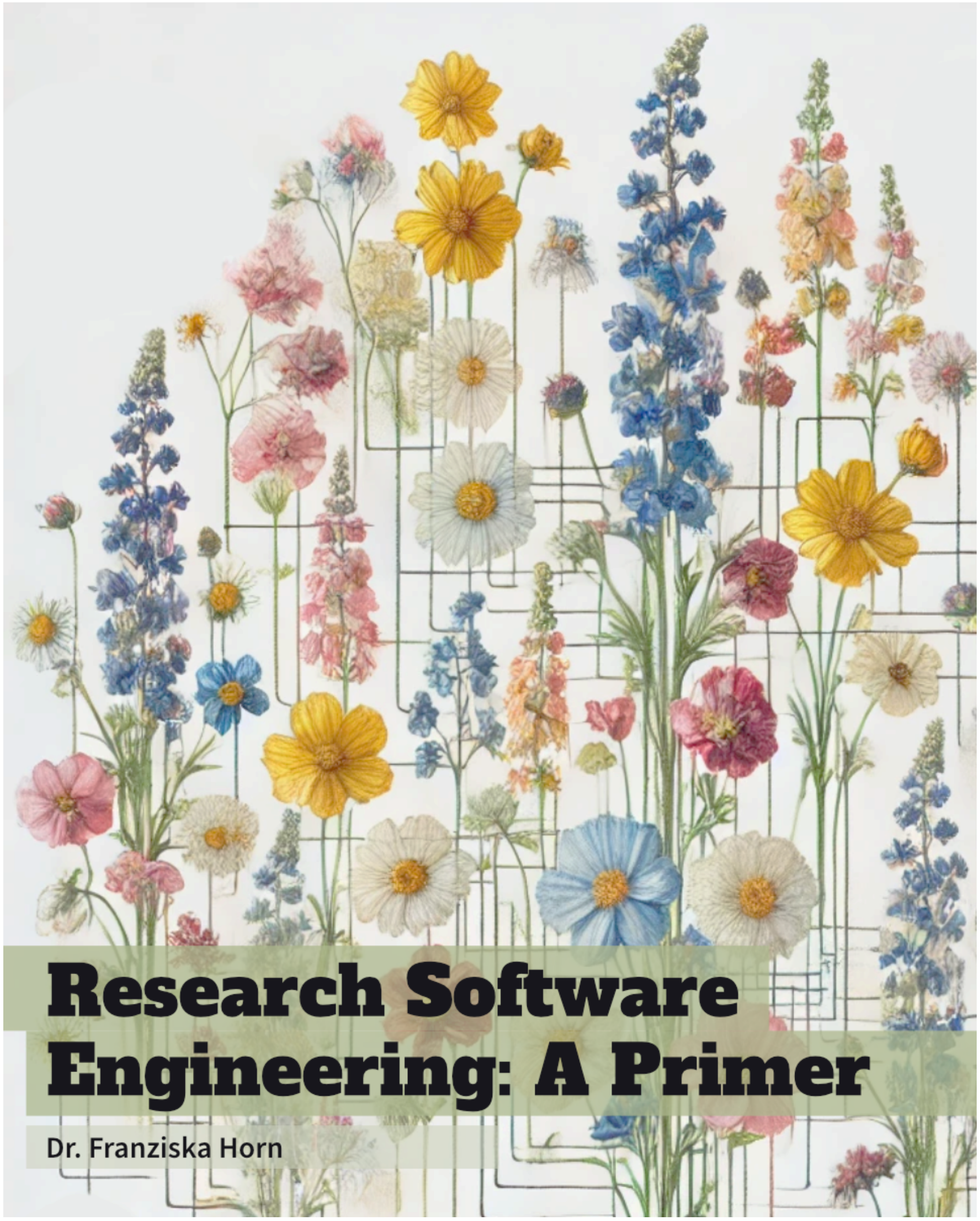


Research Software Engineering: A Primer

Dr. Franziska Horn

2025-01-01

Table of contents

Preface

This book is meant to **empower researchers to code with confidence and clarity**.

If you studied something other than computer science—especially in the natural sciences like physics, chemistry, or biology—it's likely you were never taught how to properly develop software. Yet, you're often still expected to write code as part of your daily work. Maybe you've taken a programming course like *Python for Biologists* and can put together functional scripts through trial and error (with a little help from ChatGPT). But chances are, no one ever showed you how to write well-structured, maintainable, and reusable code that could make your life—and collaborating with your colleagues—so much easier.

This book is for you if you want to:

- Write functional software more quickly
- Use a structured approach to design better programs
- Reuse your code in future projects
- Feel confident about what your scripts are doing
- Prepare your research code for production
- Share your work with pride.

Whether you're just beginning your scientific journey—perhaps working on your first major project like a master's thesis or your first paper—or you're contemplating a move from academia to industry, the practical advice in this book can guide you along the way. We will approach software design from first principles and tackle research questions with a product mindset. While the book contains some example code in Python to illustrate the concepts, the general ideas are independent of any programming language.

Software development is a craft that's best learned with the guidance of a senior colleague—someone who can show you the right tools and provide feedback through code reviews. Unfortunately, mentors with industry experience are rare in academia. While a book can't replace an apprenticeship, I hope this one gives you a head start. It's the book I wish I could have read at university and the one I always wanted to recommend to the students and junior developers I've mentored.

This is still a draft version! Please write me an email, if you have any suggestions for how this book could be improved!

Enjoy!

Acknowledgments

I would like to thank Marcel Lengert for his thoughtful feedback.

The texts in this book were partly edited and refined with the help of ChatGPT, however, all original content is my own.

How to cite

```
@book{horn2025rseprimer,
  author = {Horn, Franziska},
  title = {Research Software Engineering: A Primer},
  year = {2025},
  url = {https://franziskahorn.de/rsebook/},
}
```

1 Research Purpose

Before writing your first line of code, it's crucial to have a clear understanding of what you're trying to achieve—specifically, the purpose of your research. This clarity will not only help you reach your desired outcomes more efficiently but will also be invaluable when collaborating with others. Being able to explain your goals effectively ensures everyone is aligned and working toward the same objective.

We'll begin with an overview of common research goals and the types of data analysis needed to achieve them. Then, we'll discuss how to quantify the outcomes you're trying to achieve. Finally, we'll explore how to visually communicate your research purpose, as visual representations are often the most effective way to convey complex ideas.

1.1 Types of Research Questions

In research, your goal is to improve the status quo, whether by filling a knowledge gap or developing a new method, material, or process with better properties. Most research questions can be categorized into four broad groups, each associated with a specific type of analytics approach (Figure 1.1).

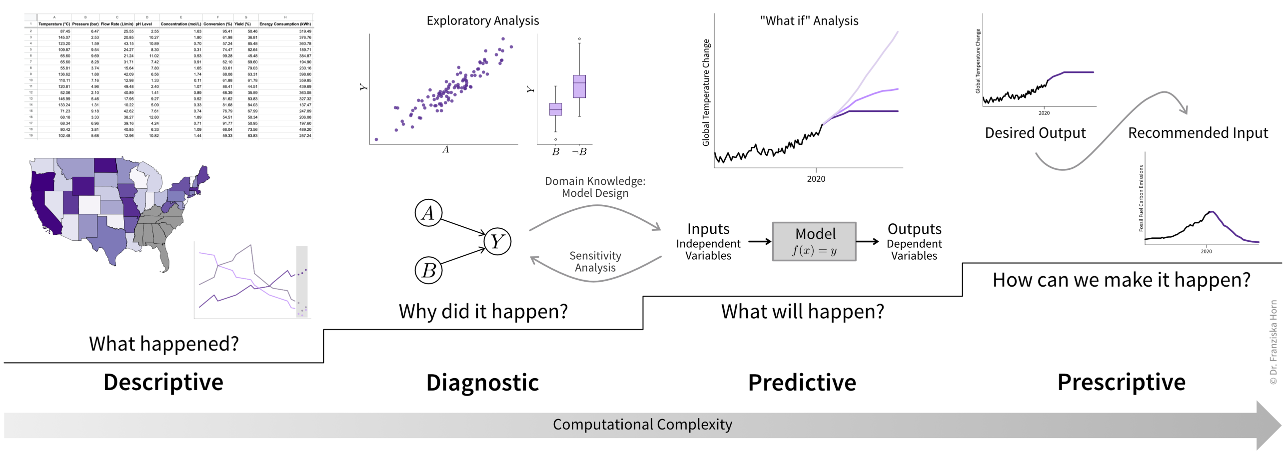


Figure 1.1: Descriptive, diagnostic, predictive, and prescriptive analytics, with increasing computational complexity and need to write custom code.

Descriptive Analytics

This approach focuses on observing and describing phenomena to establish baseline measurements or track changes over time.
Examples include:

- Identifying animal and plant species in unexplored regions of the deep ocean.
- Measuring the physical properties of a newly discovered material.
- Surveying the political views of the next generation of teenagers.

Methodology:

- Collect a large amount of data (e.g., samples or observations).
- Calculate summary statistics like averages, ranges, or standard deviations, typically using standard software tools.

Diagnostic Analytics

Here, the goal is to understand relationships between variables and uncover causal chains to explain *why* phenomena occur.
Examples include:

- Investigating how CO_2 emissions from burning fossil fuels drive global warming.
- Evaluating whether a new drug reduces symptoms and under what conditions it works best.
- Exploring how economic and social factors influence shifts toward right-wing political parties.

Methodology:

- Perform exploratory data analysis, such as looking for correlations between variables.
- Conduct statistical tests to support or refute hypotheses (e.g., comparing treatment and placebo groups).
- Design of experiments to control for external factors (e.g., randomized clinical trials).
- Build predictive models to simulate relationships. If these models match real-world observations, it suggests their assumptions correctly represent causal effects.

Predictive Analytics

This method involves building models to describe and predict relationships between independent variables (inputs) and dependent variables (outputs). These models often rely on insights from diagnostic analytics, such as which variables to include in the model and how they might interact (e.g., linear or nonlinear dependence). Despite its name, this approach is not just about predicting the future, but used to estimate unknown values in general (e.g., variables that are difficult or expensive to measure). It also includes any kind of simulation model to describe a process virtually (i.e., to conduct *in silico* experiments).
Examples include:

- Weather forecasting models.
- Digital twin of a wind turbine to simulate how much energy is generated under different conditions.
- Predicting protein folding based on amino acid sequences.

Methodology:
The key difference lies in how much domain knowledge informs the model:

- *White-box (mechanistic) models:* Based entirely on known principles, such as physical laws or experimental findings. These models are often manually designed, with parameters fitted to observed data.

- *Black-box (data-driven) models:* Derived primarily from observational data. Researchers usually test different model types (e.g., neural networks or Gaussian processes) and choose the one with the highest accuracy.

- *Gray-box (hybrid) models:* These combine mechanistic and data-driven approaches. For example, the output of a mechanistic model may serve as an input to a data-driven model, or the data-driven model may predict residuals (i.e., prediction errors) from the mechanistic model, where both outputs combined yield the final prediction.

> Resources to learn more about data-driven models
>
> If you want to learn more about how to create data-driven models and the machine learning (ML) algorithms behind them, these two free online books are highly recommended:
>
> – [9] Supervised Machine Learning for Science by Christoph Molnar & Timo Freiesleben; A fantastic introduction focused on applying black-box models in scientific research.
> – [10] A Practitioner's Guide to Machine Learning by me; A broader overview of ML methods for a variety of use cases.

After developing an accurate model, researchers can analyze its behavior (e.g., through a sensitivity analysis, which examines how outputs change with varying inputs) to gain further insights about the system (to feed back into diagnostic analytics).

Prescriptive Analytics

This approach focuses on decision-making and optimization, often using predictive models.
Examples include:

- Screening thousands of drug candidates to find those most likely to bind with a target protein.
- Optimizing reactor conditions to maximize yield while minimizing energy consumption.

Methodology:

- *Decision support:* Use models for "what-if" analyses to predict outcomes of different scenarios. For example, models can estimate the effects of limiting global warming to 2°C versus exceeding that threshold, thereby informing policy decisions.

- *Decision automation:* Use models in optimization loops to systematically test input conditions, evaluate outcomes (e.g., resulting predicted material quality), and identify the best conditions automatically.

> **!** Model accuracy is crucial
>
> These recommendations are only as good as the underlying models. Models must accurately capture causal relationships and often need to extrapolate beyond the data used to build them (e.g., for disaster simulations). Data-driven models are typically better at interpolation (predicting within known data ranges), so results should ideally be validated through additional experiments, such as testing the recommended new materials in the lab.

Together, these four types of analytics form a powerful toolkit for tackling real-world challenges: descriptive analytics provides a foundation for understanding, diagnostic analytics uncovers the causes behind observed phenomena, predictive analytics models future scenarios based on this understanding, and prescriptive analytics turns these insights into actionable solutions. Each step builds on the previous one, creating a systematic approach to answering complex questions and making informed decisions.

1.2 Evaluation Metrics

To demonstrate the impact of your work and compare your solution against existing approaches, it's crucial to define what success looks like quantitatively. Consider these common evaluation metrics to measure the outcome of your research and generate compelling results:

- **Number of samples:** This refers to the amount of data you've collected, such as whether you surveyed 100 or 10,000 people. Larger sample sizes can provide more robust and reliable results. But you also need to make sure your sample is representative of the population as a whole, i.e., to avoid sampling bias.
- **Reliability of measurements:** This evaluates the consistency of your data. For example, how much variation occurs if you repeat the same measurement, e.g., run a simulation with different random seeds. This is important as others need to be able to reproduce your results.
- **Statistical significance:** The outcome of a statistical hypothesis test, such as a p-value that indicates whether the difference in symptom reduction between the treatment and placebo groups is significant.
- **Model accuracy:** For predictive models, this includes:
  - Standard metrics like R^2 to measure how closely the model's predictions align with observational data.
  - Cross-validation scores to assess performance on new data.
  - Uncertainty estimates to understand how confident the model is in its predictions.
- **Algorithm performance:** This includes metrics like memory usage and the time required to fit a model or make predictions, and how these values change as the dataset size increases. Efficient algorithms are crucial when scaling to large datasets or handling complex simulations.
- **Key Performance Indicators (KPIs):** These are the practical measures that matter in your field. For example:
  - For a chemical process: yield, purity, energy efficiency
  - For a new material: strength, durability, cost
  - For an optimization task: convergence time, solution quality

Your evaluation typically involves multiple metrics. For example, in prescriptive analytics, you need to demonstrate both the accuracy of your model and that the recommendations generated with it led to a genuinely optimized process or product. Before starting your research, review similar work in your field to understand which metrics are standard in your community.

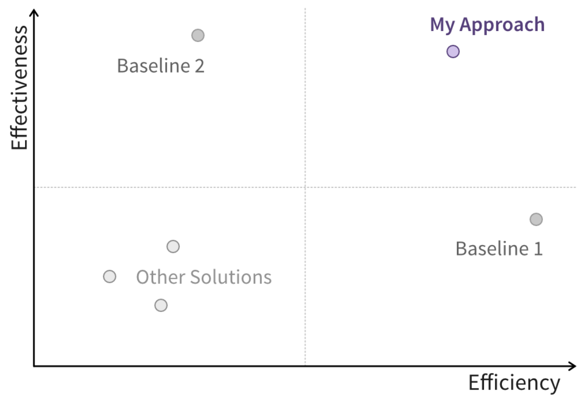


Figure 1.2: The metrics we're interested in often represent trade-offs. For example, we want a high quality product, but it should also be cheap. Or a good model accuracy, but at the same time not use excessive compute resources. Your approach might not outperform existing baselines on all metrics, but its trade-off could still be preferable.

Ideally, you should already have an idea of how existing solutions perform on these metrics (e.g., based on findings from other publications) to establish **the baseline your solution should outperform**. You'll likely need to replicate at least some of these baseline results (e.g., by reimplementing existing models) to ensure your comparisons are not influenced by external factors. But understanding where the "competition" stands can also help you identify secondary metrics where your solution could excel. For example, even if there's little room to improve model accuracy, existing solutions might be too slow to handle large datasets efficiently (Figure 1.2).[1]

These results are central to your research (and publications), and much of your code will be devoted to generating them, along with the models and simulations behind them. Clearly defining the key metrics needed to demonstrate your research's impact will help you focus your programming efforts effectively.

1.3 Draw Your Why

Whether you're collaborating with colleagues, presenting at a conference, or writing a paper—clearly communicating the problem you're solving and your proposed solution is essential.

Visual representations are particularly powerful for conveying complex ideas. One effective approach is creating **"before and after" visuals that contrast the current state of the field with your proposed improvements** (Figure 1.3).

[1]For example, currently, a lot of research aims to replace traditional mechanistic models with data-driven machine learning models, as these enable significantly faster simulations. A notable example is the AlphaFold model, which predicts protein folding from amino acid sequences—a breakthrough so impactful it was recognized with a Nobel Prize [2].

The “before” scenario might show a lack of data, an incomplete understanding of a phenomenon, poor model performance, or an inefficient process or material. The “after” scenario highlights how your research addresses these issues and improves on the current state, such as refining a predictive model or enhancing the properties of a new material.

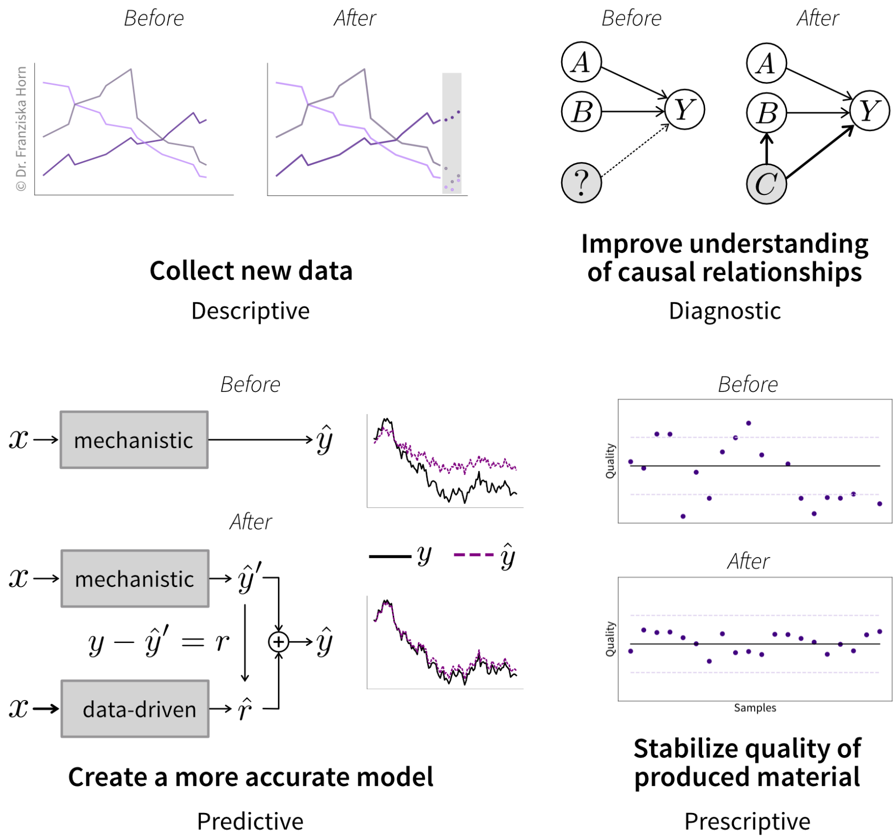


Figure 1.3: Exemplary research goals and corresponding “before and after” visuals for descriptive, diagnostic, predictive, and prescriptive analytics tasks.

At this point, your “after” scenario might be based on a hypothesis or an educated guess about what your results will look like—and that’s totally fine! The purpose of visualizing your goal is to guide your development process. Later, you can update the picture with actual results if you decide to include it in a journal publication, for example.

Of course, not all research goals are tied directly to analytics. Sometimes the main improvement is more qualitative, for example, focusing on design or functionality (Figure 1.4). Even in these cases, however, you’ll often need to demonstrate that your new approach meets or exceeds existing solutions in terms of other key performance indicators (KPIs), such as energy efficiency, speed, or quality parameters like strength or durability.

Give it a try—does the sketch help you explain your research to your family?

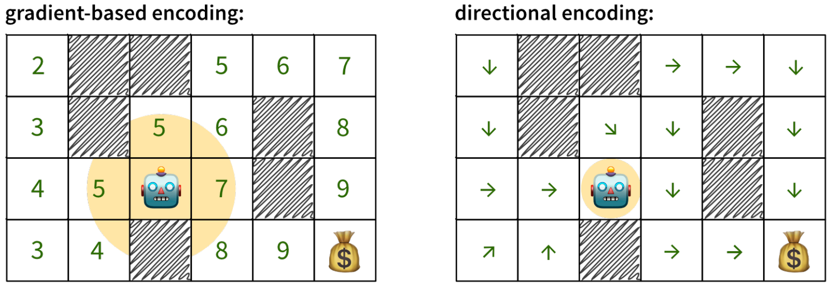


Figure 1.4: This example illustrates a task where a robot must reach its target (represented by money) as efficiently as possible. **Original approach (*left*)**: The robot relied on information encoded in the environment as expected rewards. To determine the shortest path to the target, the robot required a large sensor (shown as a yellow circle) capable of scanning nearby fields to locate the highest reward. **New approach (*right*)**: Instead of relying on reward values scattered across the environment, the optimal direction is now encoded directly in the current field. This eliminates the need for large sensors, as the robot only needs to read the value of its current position, enabling it to operate with a much smaller sensor and thereby reducing hardware costs. **Additional quantitative evaluation:** It still needs to be demonstrated that with the new approach, the robot reaches its target at least as quickly as with the original approach.

> Before you continue
>
> At this point, you should have a clear understanding of:
>
> - The problem you're trying to solve.
> - Existing solutions to this problem, i.e., the baseline you're competing against.
> - Which metrics should be used to quantify your improvement on the current state.

2 Data & Results

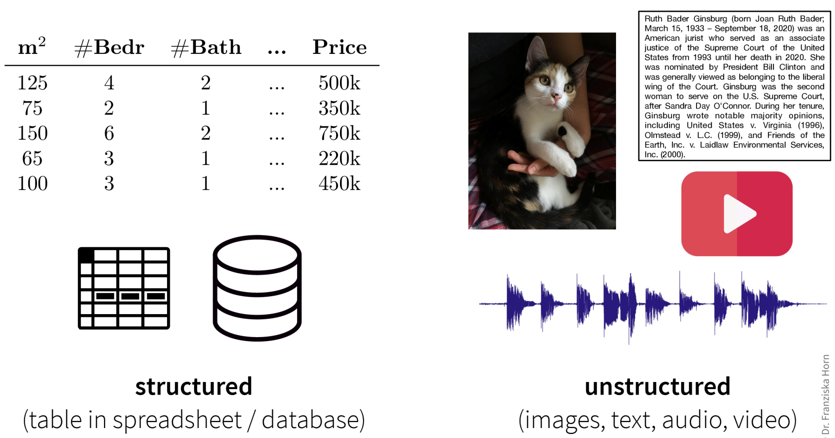
In the previous chapter, we've gained clarity on the problem you're trying to solve and how to quantify the improvements your research generates. Now it's time to dive deeper into what these results might actually look like and the data on which they are built.

2.1 Data Types

In one form or another, you're research will rely on data, both collected or generated by yourself and possibly others.

Structured vs. Unstructured Data

Data can take many forms, but one key distinction is between structured and unstructured data (Figure 2.1).



m^2	#Bedr	#Bath	...	Price
125	4	2	...	500k
75	2	1	...	350k
150	6	2	...	750k
65	3	1	...	220k
100	3	1	...	450k

Figure 2.1: Structured and unstructured data.

Structured data is organized in rows and columns, like in Excel spreadsheets, CSV files, or relational databases. Each row represents a sample or observation (a data point), while each column corresponds to a variable or measurement (e.g., temperature, pressure, household income, number of children).

Unstructured data, in contrast, lacks a predefined structure. Examples include **images, text, audio recordings, and videos**, typically stored as separate files on a computer or in the cloud. While these files might include structured metadata (e.g., timestamps, camera settings), the data content itself can vary widely—for instance, audio recordings can range from seconds to hours in length.

Structured data is often *heterogeneous*, meaning it includes variables representing different kinds of information with distinct units or scales (e.g., temperature in °C and pressure in kPa). Unstructured data tends to be *homogeneous*; for example, there's no inherent difference between one pixel and the next in an image.

> **This book focuses on structured data**
>
> Even though unstructured data is common in science (e.g., microscopy images), for simplicity, this book focuses on structured data. Furthermore, for now we'll assume that your data is stored in an Excel or CSV file, i.e., a spreadsheet with rows (samples) and columns (variables), on your computer. Later in Chapter 6, we'll discuss more advanced options for storing and accessing data, such as databases and APIs.

Programming Data Types

Each variable in your dataset (i.e., each column in your spreadsheet) is represented as a specific data type, such as:

- **Numbers** (integers for whole numbers or floats for decimals)
- **Strings** (text)
- **Boolean values** (true/false)

In programming, these are so-called *primitive data types* (as opposed to composite types, like arrays or dictionaries containing multiple values, or user-defined objects) and define how information is stored in computer memory.

> **Data types in Python**
>
> ```python
> # integer
> i = 42
> # float
> x = 4.1083
> # string
> s = "hello world!"
> # boolean
> b = False
> ```

Statistical Data Types

Even more important than how your data is stored, is understanding what your data *means*. Variables fall into two main categories:

1. **Continuous (numerical) variables** represent measurable values (e.g., temperature, height). These are usually stored as floats or integers.
2. **Discrete (categorical) variables** represent distinct options or groups (e.g., nationality, product type). These are often stored as strings, booleans, or sometimes integers.

> ⚠ Misleading data types
>
> Be cautious: a variable that looks numerical (e.g., `1`, `2`, `3`) may actually represent categories. For example, a `material_type` column with values `1`, `2`, and `3` might correspond to *aluminum*, *copper*, and *steel*, respectively. In this case, the numbers are IDs, not quantities.

Recognizing whether a variable is continuous or discrete is crucial for creating meaningful visualizations and using appropriate statistical models.

Time Series Data

Another consideration is whether your data points are linked by time. Time series data often refers to numerical data collected over time, like temperature readings or sales numbers. These datasets are usually expected to exhibit **seasonal patterns or trends** over time.

However, **nearly all datasets involve some element of time**. For example, if your dataset consists of photos, timestamps might seem unimportant, but they could reveal trends—like changes in image quality due to new equipment.

> ! Always record timestamps
>
> Always include timestamps in your data or metadata to help identify potential correlations or unexpected trends over time.

Sometimes, you may be able to collect truly time-independent data (e.g., sending a survey to 1,000 people simultaneously and they all answer within the next 10 minutes). But usually, your data collection will take longer and external factors—like an election during a longer survey period—might unintentionally affect your results. By tracking time, you can assess and adjust for such influences.

2.2 Data Analysis Results

When analyzing data, the process is typically divided into two phases:

1. **Exploratory Analysis**: This involves generating a variety of plots to gain a deeper understanding of your data, such as identifying correlations between variables. It's often a quick and dirty process to help you familiarize yourself with the dataset.
2. **Explanatory Analysis**: This focuses on creating refined, polished plots intended for communicating your findings, such as in a publication or presentation. These visuals are designed to clearly convey your results to an audience that may not be familiar with your data.

Exploratory Analysis

In this initial analysis, the goal is to get acquainted with the data, check if the trends and relationships you anticipated are present, and uncover any unexpected patterns or insights.

- Examine the **raw data**:
  - Is the dataset complete, i.e., does it contain all the variables and samples you expected?
- Examine **summary statistics** (e.g., mean, standard deviation (std), min/max values, missing value count, etc.):
  - What does each variable mean? Given your understanding of the variable, are its values in a reasonable range?
  - Are missing values encoded as NaN (Not a Number) or as 'unrealistic' numeric values (e.g., -1 while normal values are between 0 and 100)?
  - Are missing values random or systematic (e.g., in a survey rich people are less likely to answer questions about their income or specific measurements are only collected under certain conditions)? This can influence how missing values should be handled, e.g., whether it makes sense to impute them with the mean or some other specific value (e.g., zero).
- Examine the **distributions of individual (continuous) variables**:

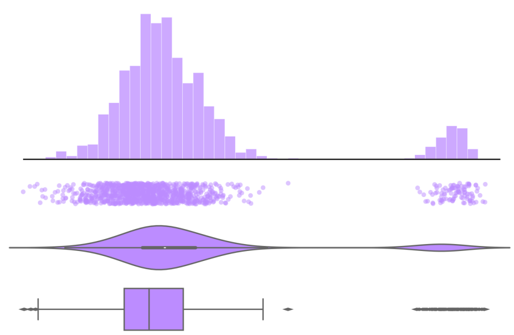


Figure 2.2: Histogram, strip plot, violin plot, box plot, and summary statistics of the same values.

  - Are there any outliers? Are these genuine edge cases or can they be ignored (e.g., due to measurement errors or wrongly encoded data)?
  - Is the data normally distributed or does the plot show multiple peaks? Is this expected?
- Examine **trends over time** (by plotting variables over time, even if you don't think your data has a meaningful time component, e.g., by lining up representative images according to their timestamps to see if there is a pattern):

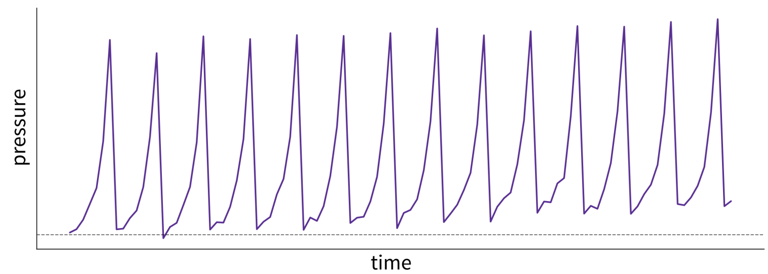


Figure 2.3: What caused these trends and what are their implications for the future? This plot shows fictitious data of the pressure in a pipe affected by fouling—that is, a buildup of unwanted material on the pipe's surface, leading to increased pressure. The pipe is cleaned at regular intervals, causing a drop in pressure. However, because the cleaning process is imperfect, the baseline pressure gradually shifts upward over time.

  - Are there time periods where the data was sampled irregularly or samples are missing? Why?
  - Are there any (gradual or sudden) data drifts over time? Are these genuine changes (e.g., due to changes in the raw materials used in the process) or artifacts (e.g., due to a malfunctioning sensor recording wrong values)?

- Examine **relationships between two variables**:

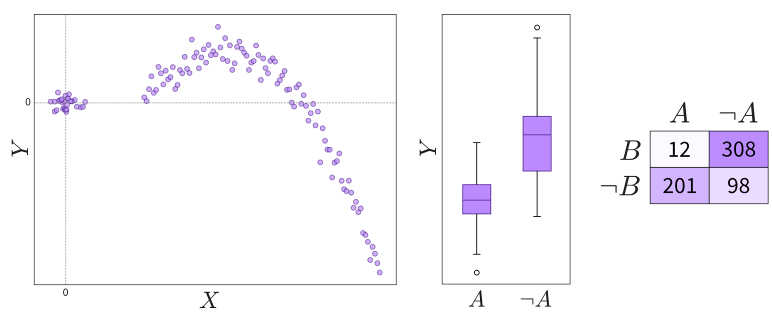


Figure 2.4: Depending on the variables' types (continuous or discrete), relationships can be shown in scatter plots, box plots, or a table. Please note that not all interesting relations between the two variables can be detected through a high correlation coefficient, so you should always check the scatter plot for details.

  - Are the observed correlations between variables expected?

- Examine **patterns in multidimensional data** (using a parallel coordinate plot):

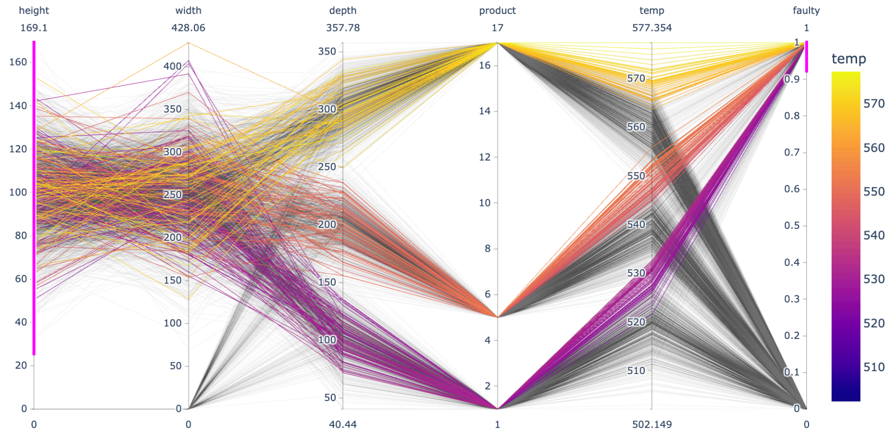


Figure 2.5: Each line in a parallel coordinate plot represents one data point, with the corresponding values for the different variables marked at the respective y-axis. The screenshot here shows an interactive plot created using the Python `plotly` library. By selecting value ranges for the different dimensions (indicated by the pink stripes), it is possible to spot interesting patterns resulting from a combination of values across multiple variables.

  – Do the observed patterns in the data match your understanding of the problem and dataset?

Explanatory Analysis

Most of the plots you create during an exploratory analysis are likely for your eyes only. Any plots you do choose to share with a broader audience—such as in a paper or presentation—should be refined to **clearly communicate your findings**. Since your audience is much less familiar with the data and likely lacks the time or interest to explore it in depth, it's essential to make your results more accessible. This process is often referred to as *explanatory analysis* [12].

> ⚠ Don't force an exploratory analysis onto your audience
>
> Don't "just show all the data" and hope that your audience will make something of it—understand what they need to answer the questions they have.

Step 1: Choose the right plot type

- Get inspired by visualization libraries (e.g., here or here), but avoid the urge to create fancy graphics; sticking with common visualizations makes it easier for the audience to correctly decode the presented information.
- Don't use 3D effects!
- Avoid pie or donut charts (angles are hard to interpret).
- Use line plots for time series data.
- Use horizontal instead of vertical bar charts for audiences that read left to right.
- Start the y-axis at 0 for area & bar charts.
- Consider using small multiples or sparklines instead of cramming too much into a single chart.

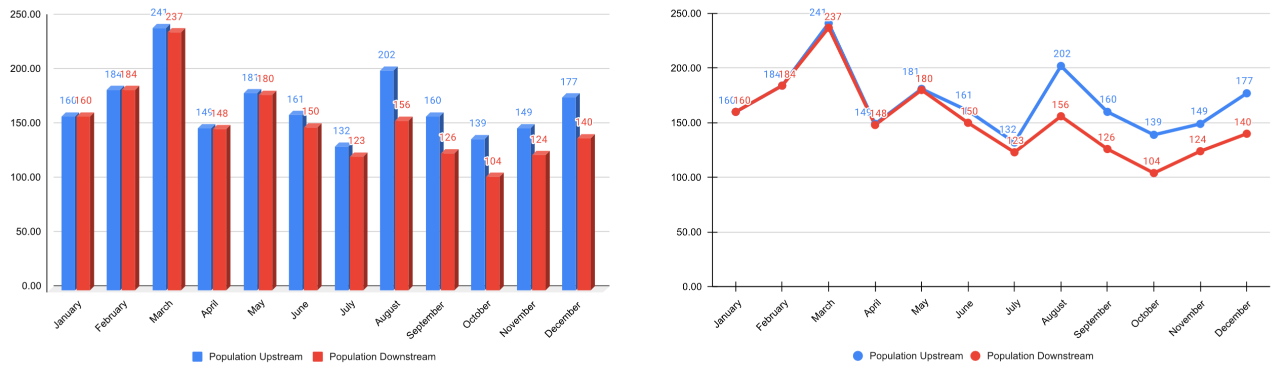


Figure 2.6: *Left:* Bar charts (especially in 3D) make it hard to compare numbers over a longer period of time. *Right:* Trends over time can be more easily detected in line charts. [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

Step 2: Cut clutter / maximize data-to-ink ratio

- Remove border.
- Remove gridlines.
- Remove data markers.
- Clean up axis labels.
- Label data directly.

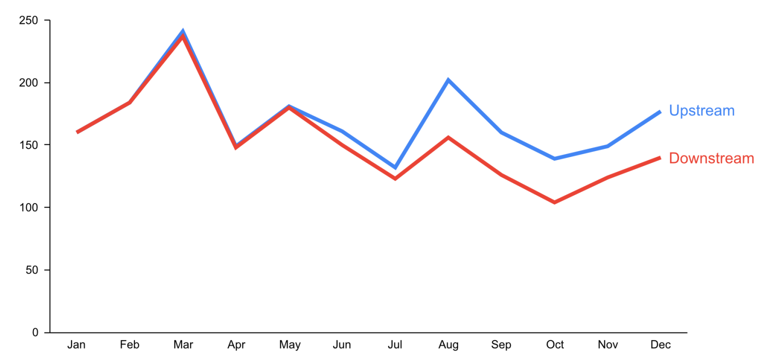


Figure 2.7: Cut clutter! [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

Step 3: Focus attention

- Start with gray, i.e., push everything in the background.
- Use pre-attentive attributes like color strategically to highlight what's most important.
- Use data labels sparingly.

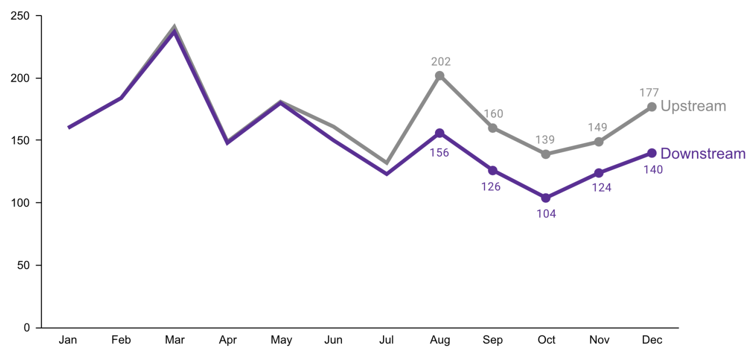


Figure 2.8: Start with gray and use pre-attentive attributes strategically to focus the audience's attention. [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

Step 4: Make data accessible

- Add context: Which values are good (goal state), which are bad (alert threshold)? Should the value be compared to another variable (e.g., actual vs. forecast)?
- Leverage consistent colors when information is spread across multiple plots (e.g., data from a certain country is always drawn in the same color).
- Annotate the plot with text explaining the main takeaways (if this is not possible, e.g., in interactive dashboards where the data keeps changing, the title can instead include the question that the plot should answer, e.g., "Is the material quality on target?").

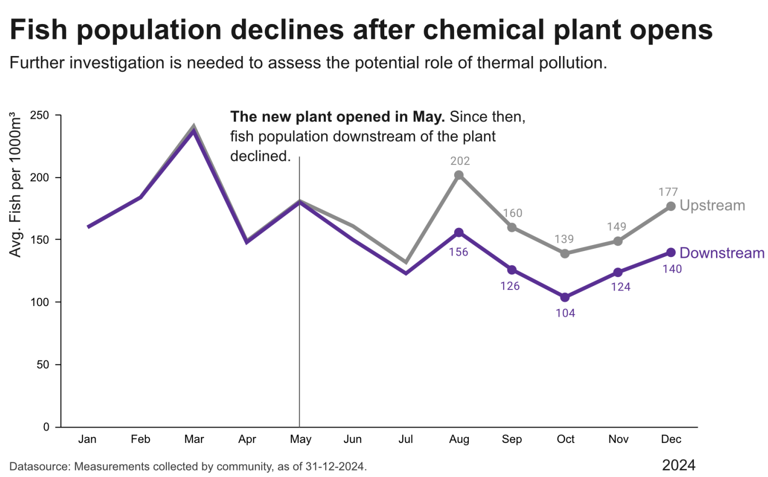


Figure 2.9: Tell a story. [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

2.3 Draw Your What

You may not have looked at your data yet—or maybe you haven't even collected it—but it's important to start with the end in mind.

In software development, a UX designer typically creates mockups of a user interface (like the screens of a mobile app) before developers begin coding. Similarly, in our case, we want to start with a clear picture of what the output of our program should look like. The difference is that, instead of users interacting with the software themselves, they'll only see the plots or tables that your program generated, maybe in a journal article.[1]

Based on your takeaways from the previous chapter—about the problem you're solving and the metrics you should use to evaluate your solution—try sketching what your final results might look like. Ask yourself: ***What figures or tables would best communicate the advantages of my approach?***

Depending on your research goals, your results might be as simple as a single number, such as a p-value or the total number of people surveyed. However, if you're reading this, you're likely tackling something that requires a more complex analysis. For example, you might compare your solution's overall performance to several baseline approaches or illustrate how your solution converges over time (Figure 2.10).

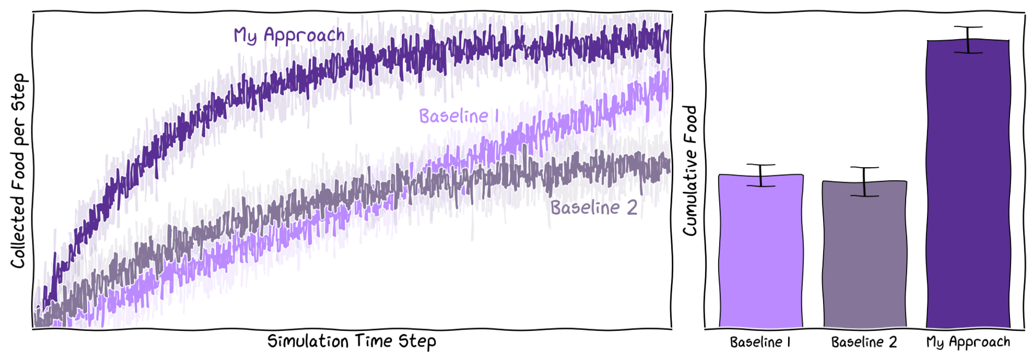


Figure 2.10: Exemplary envisioned results: The plots show the outcome of a multi-agent simulation, where 'my approach' clearly outperforms two baseline methods. In this simulation, a group of agents is tasked with locating a food source and transporting the food back to their home base piece by piece. The ideal algorithm identifies the shortest path to the food source quickly to maximize food collection. Each algorithmic approach is tested 10 times using different random seeds to evaluate reliability. The plots display the mean and standard deviations across these runs. *Left:* How quickly each algorithm converges to the shortest path (resulting in the highest number of agents delivering food back to the home base per step). *Right:* Cumulative food collected by the end of the simulation.[2]

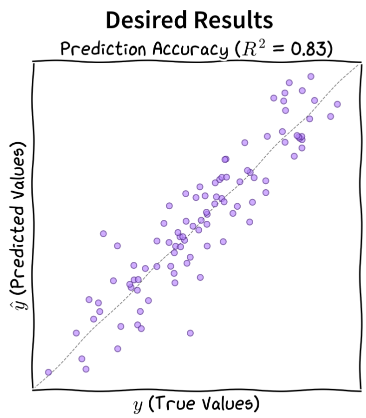
It's important to remember that your actual results might look very different from your initial sketches—they might even show that your solution performs worse than the baseline. This is completely normal. The scientific method is inherently iterative, and unexpected results are often a stepping stone to deeper understanding. By starting with a clear plan, you can generate results more efficiently and quickly pivot to a new hypothesis if needed. When your results deviate from your expectations, analyzing those differences can sharpen your intuition about the data and help you form better hypotheses in the future.

[1] A former master's student that I mentored humorously called this approach "plot-driven development," a nod to test-driven development (TDD) in software engineering, where you write a test for your function first and then implement the function to pass the test. You could even use these sketches of your results as placeholders if you're already drafting a paper or presentation.

[2] These plots and the next were generated with Python using matplotlib's `plt.xkcd()` setting and the xkcd script font. A pen and paper sketch will be sufficient for your case.

Once you've visualized the results you want, **work backward to figure out what data you need** to create them. This is especially important when you're generating the data yourself, such as through simulations. For instance, if you plan to plot how values change over time, you'll need to record variables at every time step rather than just saving the final outcome of a simulation (duh!). Similarly, if you want to report your model's accuracy (Figure 2.11), you'll need:

1. Input variables for each data point to generate predictions (= model output).
2. The actual (true) values for each data point.
3. A way to compute the overall deviation between predictions and true values, such as using an evaluation metric like R^2.



Required Data

	x1	x2	x3	x4	x5	y	y_pred
0	0.154	0.385	0.790	0.526	0.672	1.388	1.547
1	0.740	0.596	0.982	0.242	0.825	2.109	2.381
2	0.263	0.611	0.142	0.456	0.816	2.741	2.781
3	0.534	0.600	0.026	0.116	0.560	2.361	1.390
4	0.015	0.312	0.746	0.600	0.560	0.167	0.409
5	0.919	0.070	0.617	0.837	0.858	3.283	2.715
6	0.901	0.801	0.099	0.912	0.206	5.134	5.623
7	0.033	0.911	0.522	0.951	0.716	1.592	1.583
8	0.957	0.195	0.507	0.582	0.816	3.243	3.241
9	0.137	0.211	0.286	0.515	0.119	0.265	-0.046
10	0.284	0.377	0.248	0.777	0.849	1.813	2.652
11	0.606	0.403	0.180	0.672	0.125	3.015	2.643
12	0.944	0.885	0.584	0.164	0.054	3.836	4.500

Figure 2.11: Work backward from the desired results to determine what data is necessary to create them.

By working backward from your desired results to the required data, you can design your code and analysis pipeline to ensure your program delivers exactly what you need.

Before you continue

At this point, you should have a clear understanding of:

- The specific results (tables and figures) you want to create to show how your solution outperforms existing approaches (e.g., in terms of accuracy, speed, etc.).
- The underlying data needed to produce these results (e.g., what rows and columns should be in your spreadsheet).

3 Tools

Before we continue with creating your results—i.e., actually start developing software—let's take a quick tour of some tools that can make your software engineering journey smoother.

Although the code examples in this book use Python, the general principles discussed here apply to most programming languages.

3.1 Programming Languages

Different programming languages suit different needs. Here's a quick overview of some popular ones used in science and engineering:

- **R**: Commonly used for statistics, with rich functionality to create data visualizations, fit statistical models (like different types of regression), and conduct advanced statistical tests (like ANOVA). The poplar Shiny framework also makes it possible to create interactive dashboards that run as web applications.
- **MATLAB**: Once dominant in engineering, used for simulations. But due to its high licensing costs, MATLAB is being replaced more and more by Python and Julia.
- **Julia**: Gaining traction in scientific computing for its speed and modern syntax.
- **Python**: A versatile language with strong support for data science, AI, web development, and more. Its active open source community has created many popular libraries for scientific computing (numpy, scipy), machine learning (scikit-learn, TensorFlow, PyTorch), and web development (FastAPI, streamlit).

Due to its broad applicability and popularity in industry, Python is used for the examples in this book. However, you should **choose the programming language that is most popular in your field** as this will make it easier for you to find relevant resources (e.g., tailored libraries) and collaborate with colleagues.

There are plenty of great books and other resources available to teach you programming fundamentals, which is why this book focuses on higher level concepts. Going forward we'll assume that you're familiar with the basic syntax and functionality of your programming language of choice (incl. key scientific libraries). For example, to learn Python essentials, you can work through this tutorial.

3.2 Version Control

Version control is essential in software development to keep track of code changes and collaborate effectively. Think of it as a time machine that lets you revert to any version of your code or examine how it evolved.

Why Use Version Control?

- **Track changes**: See what you've modified and when, with the ability to revert if necessary.
- **Review collaborators' changes**: When working with others, reviewing their changes before they are merged with the main version of the code (in so-called pull or merge requests) ensures quality and provides opportunities to teach each other better ways of doing things.
- **Not just for code**: Version control can be used for any kind of file. While it's less effective for binary formats like images or Microsoft Word documents where you can't create a clean "diff" between two versions, you should definitely give it a try when writing your next paper in a text-based format like LaTeX.

Git

The go-to tool for version control is **Git**. While desktop clients exist, you can also use `git` directly in the terminal as a command line tool.

If you're new to Git, this beginner's guide is a great place to start.

> Essential Git Commands
>
> - `git init`: Start a new repository in the current folder.
> - `git status`: View changes.
> - `git diff`: View differences between file versions before committing.
> - `git add [file]`: Stage files for a commit.
> - `git commit -m "message"`: Save staged changes.
> - `git push`: Upload changes to a remote repository (e.g., on GitHub).
> - `git pull`: Download changes from a remote repository.
> - `git branch`: Create or list branches.
> - `git checkout [branch]`: Switch branches.
> - `git merge [branch]`: Combine branches.

By default, your repository's files are on the *main* branch. Creating a new branch is like stepping into an alternate universe where you can experiment without affecting the main timeline. **When making a major change** or adding a new feature, it's good practice to **create a new branch**, like *new-feature*, and implement your changes there. Once you're satisfied with the result, you can merge the changes back into the *main* branch.

This approach keeps the *main* branch stable and ensures you always have a working version of your code. If you decide against your new feature, you can simply abandon the branch and start fresh from *main*. By **creating a merge request** (MR) once your *new-feature* branch is ready, you or a collaborator can **review the changes** thoroughly before merging them into *main*.

To **publish your code** or **collaborate with others**, your repository (i.e., the folder under version control) can be **hosted on a platform** like:

- **GitHub**: Great for open-source projects and public personal repositories to show off your skills.
- **GitLab**: Supports self-hosting, making it ideal for organizational needs.

We strongly encourage you to publish any code related to your publications on one of these platforms to promote reproducibility of your results!

> **Data Versioning**
>
> In addition to the changes made to your code, you should also keep track of how your data is generated and transformed over time (*data lineage*). While small datasets can be included in your repository (e.g., in a separate `data/` folder), there are also more tailored tools available specifically to version your data, like DVC.

3.3 Development Environment

The program you choose for writing code directly impacts your productivity. While you can technically write code using a plain text editor (like Notepad on Windows or TextEdit on macOS), **special-purpose text editors** and **integrated development environments (IDEs)** provide a tailored experience that boosts productivity.

Text Editors

Developer-focused text editors are lightweight tools with features like syntax highlighting and extensions for basic programming tasks.
Examples include:

- **Sublime Text**: Lightweight and fast, with excellent customization through lots of plugins.
- **Atom**: Open-source and backed by GitHub (though less popular than other tools).
- **Vim** and **Emacs**: Some of the first code editors, often used as command line tools and beloved by keyboard shortcuts enthusiasts.

Terminal

When you write code in a text editor, you need a way to execute it. This is where the **terminal** comes in. A terminal, or console, lets you interact with your computer through the **command line**, using text-based commands. Think of it like stepping back to the 1970s—or like being one of those cool hackers you see on TV.

On macOS and Linux, a terminal app is already preinstalled. On Windows, different options exist to install a Unix-like terminal, like the Windows Terminal. Inside the terminal, there's a **shell**: the actual program that processes the commands you type. The most common shells on Unix systems are `bash` and `zsh`, which are quite similar. For this book, we'll assume you're using one of these.

With the shell, you can navigate your computer's file system and run programs through their command-line interface (CLI). Try it out!

Basic Shell Commands

Follow along by typing these commands into your terminal. In parallel, you can watch your normal file browser to see files and folders appear or disappear as you go.

- `pwd`: Print the current working directory—this shows the path to where you opened the terminal.
- `ls`: List files and directories in the current location. Use `ls -la` for more details, including hidden files (like `.gitignore`).
- `cd path/to/folder`: Change directory to the specified path. Tips: Use tab to autocomplete names. If the path starts with `/`, it's absolute (from the file system's root). If it starts with `~/`, it's relative to your home directory. Use `..` to move up one folder.
- `mkdir new_folder`: Create a new directory named `new_folder`.
- `touch new_file.txt`: Create an empty file named `new_file.txt`.
- `cp new_file.txt copied_file.txt`: Copy `new_file.txt` to `copied_file.txt`. Use `mv` instead of `cp` to move or rename files.
- `rm new_file.txt`: Delete `new_file.txt`. Add `-r` to delete directories. But be careful: files deleted this way bypass the trash and are gone for good, so double-check before hitting enter!

You can also run other CLI programs in the terminal, like using the `git` commands described earlier.
A Python script can be executed with `python script.py` (assuming the script is in your current directory).

Not all CLI programs mentioned in this book will be preinstalled on your machine. Linux systems already come with a command-line package manager (like `apt` on Ubuntu), which can be used to install other tools. A popular package manager for macOS is `brew`, while for Windows you can use `winget`.

Once you get comfortable with your shell, you can also create **shell scripts** (files with a `.sh` extension) to automate tasks and handle more complex workflows. These scripts can include conditionals, loops, and other programming constructs. For more information on bash scripting, check out this resource.

Full IDEs

Integrated Development Environments (IDEs) combine all the tools you need in one place—file browser, editor, terminal, Git support, debugger, and more. They are ideal for larger projects and provide support for more complex tasks, like renaming variables across multiple files when you're refactoring your code.
Examples include:

- **VS Code**: Minimalist by default but highly customizable with plugins, making it suitable for everything from basic editing to full-scale development.
- **JetBrains IDEs** (e.g., PyCharm): IDEs tailored to the needs of specific programming languages with very advanced features. You need to purchase a license to use the full version, but for many IDEs there is also a free community edition available.

- **JupyterLab**: An extension of Jupyter notebooks (see below), popular for data science and exploratory coding.
- **RStudio**: Tailored for R programming, with excellent support for data visualization, markdown reporting, and reproducible research workflows.
- **MATLAB**: The MATLAB programming language and IDE are virtually synonymous. However, its rich feature set comes with steep licensing fees.

Jupyter Notebooks

Jupyter notebooks are a unique format that lets you **mix code, output (like plots), and explanatory text** in one document. The name *Jupyter* is derived from Julia, Python, and R, the programming languages for which the notebook format, and later the JupyterLab IDE, were created. The IDE itself runs inside your web browser.

Notebooks are great for exploratory data analysis and to create reproducible reports. However, since the notebooks themselves are composed of individual interactive cells that can be executed in any order, developing in notebooks often becomes messy quickly. We recommend that you keep the main logic and reusable functions in separate scrips or libraries and primarily use notebooks to create plots and other results. It is also good practice once you're finished to restart the kernel and run your notebook again from top to bottom to make sure everything still works and you're not relying on variables that were defined in now-deleted cells, for example.

> Notebooks as text files
>
> Jupyter notebooks, stored as files ending in `.ipynb`, are internally represented as JSON documents. If you have your notebooks under version control (which you should), you'll notice that the diffs between versions look quite bloated. But do not despair! Tools like Jupytext can convert notebooks into plain text without loss of functionality.

> Parameterize notebooks
>
> If you want to execute the same notebook with multiple different parameter settings (e.g., create the same plots for different model configurations), have a look at papermill.

In addition to the original JupyterLab IDE and notebooks that you install on your computer, there are also free cloud-based options available, such as Google Colab, which even gives you free compute time on GPUs.

3.4 Reproducible Setups

"It works on my machine" isn't good enough for science. Reproducibility means your results can be replicated by others (and by you a few months later when the reviewers of your paper request changes to your experiments). The first step to achieve this is to **manage your dependencies** (i.e., external libraries used by your code) to ensure the environment in which your code is executed is identical for everyone that runs your code, every time. This can be done using **virtual environments**, or, if you want to go even further, **containers** like Docker, which will be discussed in Chapter 6.

Virtual Environments in Python with `poetry`

Virtual environments isolate your project's dependencies, thereby ensuring consistency. For Python, a common tool to do this is poetry. It tracks the libraries and their versions in a `pyproject.toml` like this:

```
[tool.poetry]
name = "example-project"
version = "0.1.0"
description = "A sample Python project"
authors = ["Your Name <youremail@example.com>"]

[tool.poetry.dependencies]
python = "^3.9"
requests = "^2.26.0"  # external libraries incl. versions

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Basic commands:

- `poetry new example-project`: Create a new project (folder incl. `pyproject.toml` file).
- `poetry add [package]`: Add a dependency (can also be done directly in the file).
- `poetry install`: Install all dependencies.
- `poetry shell`: Activate the virtual environment.

Handling Randomness

Your program will often depend on randomly sampled values, for example, when defining the initial conditions for a simulation or initializing a model before it is fitted to data (like a neural network). To ensure that your experiments can be reproduced, it is important that you always **set a random seed** at the beginning of your program so the random number generator starts from a consistent state.

Setting Random Seeds in Python

At the beginning of your script, set a random seed (depending on the library that you're using this can vary):

```python
import random
import numpy as np

random.seed(42)
np.random.seed(42)
```

To get a better idea of how much your results depend on the random initialization and therefore how robust they are, it is advisable to always **run your code with multiple random seeds and compare the results** (e.g., compute the mean and standard deviation of the outcomes of different runs like in Figure 2.10).

> **Random state at startup**
>
> Depending on the programming language that you're using, if you run a script without executing any other code before, the random number generator may or may not always start in the same state. This means, if you don't set a random seed and, for example, run your script ten times from scratch, you may always receive the same result even though the results would differ if the code was run under different circumstances. To avoid surprises, you should always explicitly set the random seed to have more control over the results.

> **Hardware differences**
>
> If your code is run on very different hardware, e.g., a CPU vs. a GPU (graphics card, used to train neural network models, for example), despite setting a random seed, your results might still differ slightly. This is due to how the different architectures internally represent float values, i.e., with what precision the numbers are stored in memory.

3.5 Clean and Consistent Code

Especially when working together with others, it can be helpful to **follow to a style guide to produce clean and consistent code**. Google published their style guides for multiple programming languages, which is a great resource and adhering to these rules will also help you to avoid common sources of bugs.

Formatters & Linters

Since programmers are often rather lazy, they developed tools that automatically fix your code to implement these rules where possible:

- **Formatters** rewrite code to follow a consistent style (e.g., add whitespace after commas).
- **Linters** analyze code for errors, inefficiencies, and deviations from best practices.

> **Formatter & Linter in Python: `ruff`**
>
> `ruff` is a (super fast) formatter and linter for Python, written in Rust. You can install it via `pip` and configure it in the same `pyproject.toml` file that we also used for `poetry`. Then run it over you code like this:

```
ruff check         # see which errors the linter finds
ruff check --fix   # automatically fix errors where possible
ruff format        # automatically format the code
```

> You'll probably want to add exceptions for some of the errors that the linter checks for in your `pyproject.toml` file as `ruff` is quite strict.

It is important to have the configuration for your formatter and linter under version control as well, so that all collaborators use the same settings and you avoid unnecessary changes (and bloated diffs in merge requests) when different people format the code.

Pre-commit Hooks

In the heat of the moment, you might forget to run the formatter and linter over your code before committing your changes. To **avoid accidentally checking messy code into your repository**, you can configure so-called "pre-commit hooks". Pre-commit hooks catch issues automatically by enforcing coding standards before committing or pushing code with git.

> Setting up pre-commit hooks
>
> First, you need to install pre-commit hooks, e.g., through Python's package manger `pip`:
>
> ```
> pip install pre-commit
> ```
>
> Then configure it in a file named `.pre-commit-config.yaml` (here done for `ruff`):
>
> ```
> repos:
> - repo: https://github.com/pre-commit/pre-commit-hooks
>   rev: v2.3.0
>   hooks:
>     - id: check-yaml
>     - id: end-of-file-fixer
>     - id: trailing-whitespace
> - repo: https://github.com/astral-sh/ruff-pre-commit
>   # Ruff version.
>   rev: v0.8.3
>   hooks:
>     # Run the linter.
>     - id: ruff
>       args: [ --fix ]
>     # Run the formatter.
>     - id: ruff-format
> ```
>
> Then install the git hook scripts from the config file:
>
> ```
> pre-commit install
> ```
>
> Now the configured hooks will be run on all changed files when you try to commit them and you can only proceed if all checks pass.

To catch any style inconsistencies after the code was pushed to your remote repository (e.g., in case one of your collaborators has not installed the pre-commit hooks), you can also add these checks to your CI/CD pipeline (see Chapter 6).

3.6 Putting It All Together

When you set up all these tools, your repository should now look something like this (see here for more details; setup for programming languages other than Python will differ slightly):

```
project-name/
  .gitignore              # Exclude unnecessary files from version control
  README.md               # Describe the project purpose and usage
  pre-commit-config.yaml  # Pre-commit hook setup
  pyproject.toml          # Python dependencies and configs
  data/                   # Store (small) datasets
  notebooks/              # For exploratory analysis
  src/                    # Core source code
  tests/                  # Unit tests
```

A clean project structure makes it easier to maintain your code.

> Before you continue
>
> At this point, you should have a clear understanding of:
>
> - How to set up your development environment to code efficiently.
> - How to host your version-controlled repository on a platform like GitHub or GitLab, complete with pre-commit hooks to ensure well-formatted code.
> - The fundamental syntax of your programming language of choice (incl. key scientific libraries) to get started.

4 Software Design

Now that your code repository is set up, are you itching to start programming? Hold on for a moment!

One of the most **common missteps** I've seen junior developers take is **jumping straight into coding without first thinking through what they actually want to build**. Imagine trying to construct a house by just laying bricks without consulting an architect first—halfway through you'd probably realize the walls don't align, and you forgot the plumbing for the kitchen. You'd have to tear it down and start over! To avoid this fate for your software, it's essential to make a plan and design the final outcome first. It doesn't have to be perfect—a quick sketch on paper will do. And you'll likely need to adapt as you go (which is fine since we'll design with flexibility in mind!). But the more thought you put into planning, the smoother and faster execution will be.

To make sure your designs will be worthy of implementation, this chapter also covers key paradigms and best practices that will help you create **clean, maintainable code that's easy to extend and reuse** in future projects.

4.1 Avoid Complexity

A common acronym in software engineering is **KISS** - "keep it simple, stupid!". While this may be well-intentioned advice, it can only apply to individual parts of your code, as a full-fledged software is a system that consists of multiple components that interact with each other. Here, the best you can hope for is *complicated*, i.e., to avoid complexity (Figure 4.1).

A **complex** system includes many elements that interact with each other in ways that are not easily traceable or for humans to comprehend. You might change something in one place and suddenly, something far on the other side breaks. That's not good in software. We should always design for change, since inevitably, there is something we need to adapt or extend and when this happens, we want to be confident in what we're doing and not afraid that our change will break something else or have unintended consequences that we're not aware of. No one likes bugs.

This is why we want a **complicated** system: it still includes a lot of elements, but they are grouped in components, neat little subsystems, so the whole can be taken apart and each part can be understood on its own while the whole can also be understood without understanding each individual component in detail. Ideally, this means our system consists of a neat hierarchy of components at different levels of abstraction (Figure 4.2). To understand the code on one level of this hierarchy, we only need to understand what the components one level below are doing to get an idea of the purpose of the code. For example, to understand what is happening in `plot_results`, it is enough to know that there is a scatter plot created and we don't need to know the details of how this plot is created, such as that it requires the computation of R^2. By decomposing code in such a way, we reduce the **cognitive load** that is required to understand what the code is doing.

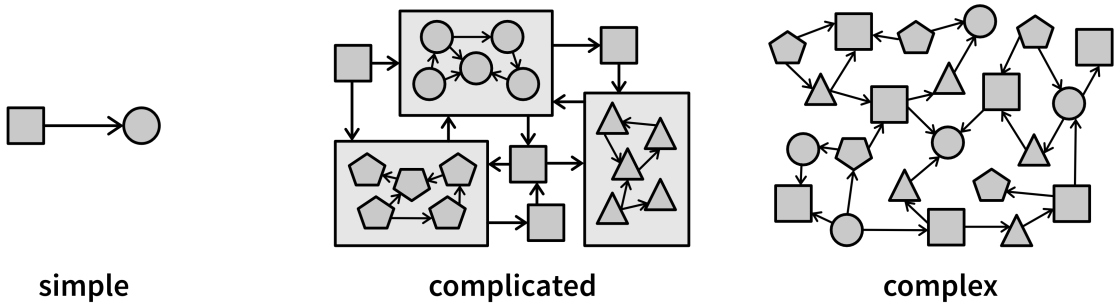


Figure 4.1: (Software) Systems can be of different complexity. A script or function that linearly executes a set of steps could be considered simple. But most software programs are (at least) complicated: they consist of multiple components that interact with each other. However, these can still be broken down into manageable subsystems, which makes it possible to understand the system as a whole. A complex system, in computer science referred to as "spaghetti code" or a "big ball of mud" [5], contains many individual elements and interactions between them – when you change something on one end it is unclear how this will affect the other pieces as it is difficult to understand how all the elements come together. (Figure adapted from [13])

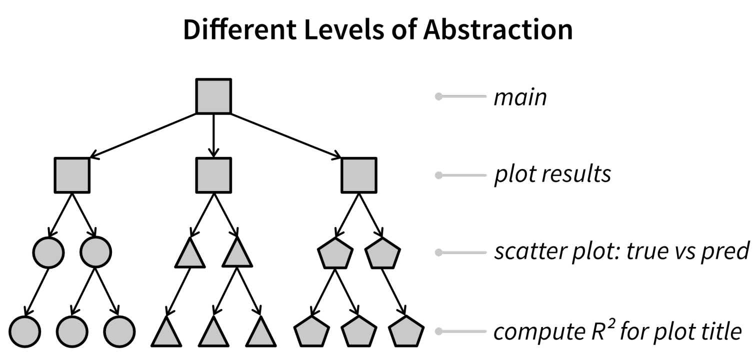


Figure 4.2: Complicated systems in software design can usually be represented as hierarchies that show different levels of abstraction, e.g., in this case for plotting the results of a predictive model, i.e., creating a scatter plot that shows the true vs. predicted values together with the R^2 value that indicates the overall performance of the model.

Elements of Software Systems

Before we continue, lets take a quick look at what the fundamental building blocks of our software systems are. To keep things simple, we distinguish between:

- **Variables:** Used to store data values, either primitive (like integers or strings, like we saw in Chapter 2) or composite, like lists and dictionaries, which can grow arbitrarily complex in some languages (e.g., a list can contain multiple dictionaries that themselves contain lists etc.).

```
# primitive data type: float
x = 4.1083
# composite data type: list
my_list = ["hello", 42, x]
# composite data type: dict
my_dict = {
    "key1": "hello",
    "key2": 42,
    "key3": my_list,
}
```

- **Functions:** Used to calculate something and/or perform an action, usually given some input arguments. We distinguish between pure and impure functions [14]. A **pure function** is similar to a mathematical function $f : x \rightarrow y$ that computes and returns y given some x. **Impure functions** have so-called "side effects", i.e., they perform some action that has effects that are visible outside of the function itself, e.g., because they create or modify a file. They also don't necessarily have a return value.

```
def add(a, b):
    # pure: a simple calculation
    return a + b

def create_plot(x, y):
    # impure: create and save a plot
    plt.plot(x, y)
    plt.savefig("my_figure.png")
```

- **Objects:** They are basically a combination of variables and functions, i.e., specific constructs that store data values and have functions that usually access and modify these values. This comes in handy if you want to group multiple variables in one place because they logically belong together (e.g., the parameters used to configure a simulation model) and you want to use them store the results from functions.

```
class MyModel:

  def __init__(self, param1, param2):
      # initialize values from input arguments
      self.param1 = param1
      self.param2 = param2
      # create additional variables
```

```
        self.results = []

    def run_simulation():
        # do something
        self.result = [1, 2, 3]
```

System Boundaries

To avoid creating a complex mess, it is very important that we are clear on our (sub)system's boundaries, i.e., what is in and what is out of our code's scope. Adhering to clear boundaries makes it very easy to change our code. Unfortunately, though, establishing clear boundaries is easier said than done.

Let's start with a very simple example to illustrate the concept of scope:

```
def n_times_x(x, n):
    result = 0
    for i in range(n):
        result += x
    return result

if __name__ == '__main__':
    new_x = 3
    new_n = 5
    # call our function with some values
    my_result = n_times_x(new_x, new_n)
    print(my_result)  # should show 15
```

Our code of interest (i.e., the subsystem we're looking at) is the function `n_times_x`. Inside the scope of this function with have the variables `x`, `n`, `i`, and `result`, i.e., when we can refer to them by name and when we execute the code it is clear what we're referring to. Outside of the scope of the function, where it is called in the program's `__main__` function, we have the variables `new_x`, `new_n`, and `my_result`. If we tried to access any of the internal variables from `n_times_x` here, we would get an error, since these variables are hidden inside the scope of `n_times_x`.

> **Note**
>
> While under some circumstances, depending on how you structure your code, the "inside code" could access the "outside code"'s variables, it is best to avoid this, i.e., try to have a clean separation between what is going on inside and outside of your code. Ideally, your code should only be concerned about what is going on inside your code and not depend on anything that exists outside of it. Therefore, if it needs access to any variables, you should just pass them as input arguments explicitly to make it clear what values your code uses.

This brings us to the boundaries, i.e., where "inside" and "outside" code meet. In software development, the boundary of your "inside code" is also referred to as its **interface**. This defines how you interact with the code and it can be seen as a contract with the outside world for how to use your code. For a pure function, like in our example, the interface is the function's signature, i.e.,

- The function name (`n_times_x`).
- The input arguments (`x` and `n`).
- The return values (`result`).

As long as we keep this interface the same, i.e., the function still calculates the same thing, we're free to change whatever we want inside the function itself. For example, we could change the ridiculously inefficient implementation to use a proper multiplication:

```
def n_times_x(x, n):
    result = n * x
    return result
```

And we don't have to tell anyone about this change, since we kept the interface of the function the same and none of the outside code was aware or depended upon what was going on inside our function. This is the beauty of clear boundaries.

On the other hand, if you decide that you don't like your function's name, changing this means that everywhere else where your function is used you also need to change the name. When you use an IDE to code, it probably has a feature to refactor your code where you can indicate that you want to change the name and then it checks all the files in your project for where this function is called and then changes the name there as well. However, if you're writing a library that is used in multiple place outside your current project, it will be very difficult to identify all the places where this function is used and notify the respective people to change the name. In practice this requires a deprecation process where the old interface is slowly phased out. This is why it really pays of to define good interfaces that remain stable for a long time.

Go deep

Powerful code has narrow interfaces with a deep implementation, i.e., the code does meaningful computations without exposing too much of its internals. For example, a narrow and deep function would maybe have two input arguments but extend over ten lines of code to do a complex calculation. When you split your code up into reusable functions, make sure that you don't split it up so much that you end up with a function that takes six input arguments but then only computes something on one line. Instead, try to create meaningful units of code that help you to hide complicated logic behind a simple interface and thereby make your code easier to change.

Pure functions with clean boundaries help us to write easily understandable code without any surprises. They are also easy to test, since they do not depend on anything besides what is passed as their input arguments, so no matter how many times you call a pure function with the same inputs, you're always going to get the same output. Your code gets more complex with **impure functions**: by definition, they have side effects and therefore interact and rely on the outside world. For example, they could write results to a file (e.g., create a plot) or read values from a database. This also means that if you run the code twice, you might get different results because something in the outside world has changed, e.g., your code crashes because the results file already exist or the output is different because someone has changed the values in the database. This can make it harder to predict how changes to your code or somewhere else will affect the system as a whole, for example, because you don't know who else is accessing the same resources that your code uses.

As a best practice, you should always try to encapsulate as much critical logic as possible in pure functions, as this makes your code easier to understand and test. For example:

```python
def pure_function(inputs):
    # do something without side effects
    output = ...
    return output

def impure_function():
    # read from outside file/database
    data = ...
    # do the main calculations
    result = pure_function(data)
    # write to outside file/database
    result.to_csv("result.csv")
```

Extend your scope with objects

Sometimes, your code might depend on so many variables that you don't want to pass all of these to a function. Especially when all of these variables and functionality are logically related, they should be grouped together into a class. A class, or an object, with is one concrete instantiation of a class, is great to create some extra scope for your functions.

Since we want clean boundaries and ideally a narrow interface, it is important to distinguish between *public* and *private* attributes and methods of a class. Everything about a class that is public is part of its interface, i.e., the contract with the outside world, which means changing these later can cause extra work or issues elsewhere. Private attributes and methods are only for internal use and can therefore be changed more easily.

> **i** Public and Private in different programming languages
>
> Depending on the programming language that you use, there might be more access levels than *public* and *private*. For example, in Java there also exists *protected* and *package*. For our purposes it is sufficient to distinguish between what should be accessible internally (for your code only) and externally (part of the contract with the outside world).
> In Python, nothing is really private as it is assumed that the programmer knows what she is doing when she accesses what she wants. By convention, variables and functions that are prefixed with `_` or `__` are for internal use only and you should only use them at your own risk. Therefore, it also makes sense to prefix all attributes and methods that you don't want anyone else to mess with with underscores. While this does not provide any access guarantees, at least who ever uses these variables will be warned that they may change without notice.

4.2 Draw Your How

Now it's time for you to draw your design.

But keep it simple—you don't have to overengineer your design to account for every possibility. If you want to build a one family home, design a one family home. Of course, it can't hurt to think ahead a little bit if you already know that some changes are likely to come ("How will the rooms change as the kids move out?"), but there is no value in trying to plan ahead for everything ("What if the daughter wants to take over the house and transform it into an office for her 100+ people company?").

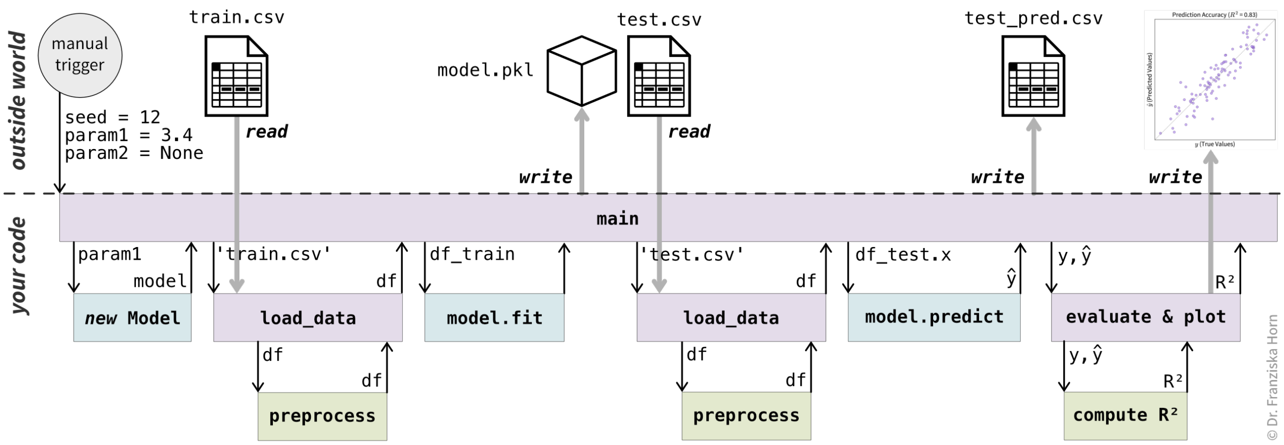


Figure 4.3: green boxes represent pure functions, blue boxes class methods, and purple boxes impure functions with side effects (reading or writing to external files)

4.3 Make a Plan

Before you continue

At this point, you should have a clear understanding of:

- How to design your program by building upon loosely coupled, reusable components.
- What steps your code entails.

5 Implementation

Now that you have a plan, it's finally time to get started with the implementation.

5.1 From Design to Code: Fill in the Blanks

Armed with your design from the last chapter, you can now **translate your sketch into a code skeleton**. Start by outlining the functions, place calls to them where needed, and add comments for any steps you'll figure out later. For example, the design from Figure 4.3 could result in the following draft:

```
import numpy as np
import pandas as pd

class Model:
    def __init__(self, param1):
        self.param1 = param1

    def fit(self, x, y):
        pass

    def predict(self, x):
        y = ...
        return y

def preprocess(df):
    df = ...
    return df

def load_data(file_name):
    df = pd.read_csv(file_name)
    df = preprocess(df)
    return df

def compute_R2(y, y_pred):
    R2 = ...
    return R2

def evaluate_and_plot(y, y_pred):
    R2 = compute_R2(y, y_pred)
```

```
    # ... create and save plot ...
    return R2

if __name__ == '__main__':
    # script is called as `python script.py seed param1 [param2]`
    seed, param1, param2 = ...
    np.random.seed(seed)
    model = Model(param1)
    df_train = load_data("train.csv")
    model.fit(df_train.x, df_train.y)
    df_test = load_data("test.csv")
    y_pred = model.predict(df_test.x)
    R2 = evaluate_and_plot(df_test.y, y_pred)
```

> Order of functions
>
> Your script likely includes multiple functions, so you'll need to decide their order from top to bottom. Since scripts typically start with imports (e.g., of libraries like `numpy`) and end with a `main` function, personally I prefer to put more general functions (i.e., the ones that are at the lower levels of abstraction in your call hierarchy and that only rely on external dependencies) towards the top of the file. This ensures that, as you read the script from top to bottom, **each function depends only on what was defined before it**. Maintaining this order avoids circular dependencies and encourages you to write reusable, modular functions that serve as building blocks for the code that follows.

Once your skeleton stands, you "only" need to **fill in the details**, which is a lot less intimidating than facing a blank page. Plus, since you started with a thoughtful design, your final program is more likely to be well-structured and easy to understand. Compare this to writing code on the fly, where decisions about functions are often made haphazardly—you'll appreciate the difference.

> Using AI Code Generators
>
> AI assistants like ChatGPT or GitHub Copilot can be helpful tools when writing code, especially at the level of individual functions. However, remember that these tools only reproduce patterns from their training data, which includes both good and bad code. As a result, the code they generate may not always be optimal. For instance, they might use inefficient for-loops instead of more elegant matrix operations. Similarly, support for less popular programming languages may be subpar.
> To get better results, consider crafting prompts like: *"You are a senior Python developer with 10 years of experience writing efficient, edge-case-aware code. Write a function …"*

Minimum Viable Results

In product development, there's a concept called the **Minimum Viable Product (MVP)**. This refers to the simplest version of a product that still provides value to users. The MVP serves as a prototype to gather feedback on whether the product meets user needs and to identify which features are truly essential. By iterating quickly and testing hypotheses, teams can increase the odds of creating a successful product that people will actually pay for.

This approach also has motivational benefits. Seeing something functional—even if basic—early on makes it easier to stay engaged. It's far better than toiling for months without tangible results. We recommend applying this mindset to your research software development by **starting with a script that generates "Minimum Viable Results."**

This means creating a program that produces outputs resembling your final results, like plots or tables, but using placeholder data instead of actual values. For instance:

- If your goal is to build a prediction model, start with one that simply predicts the mean of the observed data.
- If you're developing a simulation, begin with random outputs, such as a random walk.

By starting with Minimum Viable Results, you can **test your code end-to-end** early on, see tangible progress, and **iteratively improve** from there.

This approach also serves as a **"stupid baseline"**—a simple, easy-to-beat reference point for your final method. It's a sanity check: if your sophisticated solution can't outperform this baseline, something's off.

Breaking Code into Modules

Starting a new project often begins with all your code in a single script or notebook. This is fine for quick and small tasks, but as your project grows, keeping everything in one file becomes messy and overwhelming. To keep your code organized and easier to understand, it's a good idea to move functionality into separate files, also called (sub)modules. **Separating code into modules makes your project easier to navigate, test, and reuse.**

A typical first step is splitting the main logic of your analysis (`main.py`) from general-purpose helper functions (`utils.py`). Over time, as `utils.py` expands, you'll notice clusters of related functionality that can be moved into their own files, such as `preprocessing.py`, `models.py`, or `plot_results.py`. This modular approach naturally leads to a clean directory structure, which might look like this for a larger Python project:[1]

```
src/
  my-package/
      __init__.py
      main.py
      models/
          __init__.py
          baseline_a.py
          baseline_b.py
          my_model.py
```

[1]The `__init__.py` file is needed to turn a directory into a package from which other scripts can import functionality. Usually, the file is completely empty.

```
    utils/
        __init__.py
        preprocessing.py
        plot_results.py
```

In `main.py`, you can import the relevant classes and functions from these modules to keep the main script clean and focused:

```python
from models.my_model import MyModel
from utils import preprocessing

if __name__ == '__main__':
    # steps that will be executed when running `python main.py`
    model = MyModel()
```

> Keep helper functions separate
>
> Always **separate reusable helper functions from the main executable code**. This also means that files like `utils/preprocessing.py` should not include a *main* function, as they are not standalone scripts. Instead, these modules provide functionality that can be imported by other scripts—just like external dependencies such as `numpy`.
> As you tackle more projects, you may develop a set of functions that are so versatile and useful that you find yourself reusing them across multiple projects. At that point, you might consider packaging them as your own open-source library, allowing others to install and use it just like any other external library.

Keep It Compact

When writing code, aim to achieve your goals while using **as little screen space as possible**—this applies to both the number of lines and their length.

> Tips to create compact, reusable code
>
> - **Avoid duplication**: Instead of copying and pasting code in multiple places, consolidate it into a reusable function to save lines.
> - **Prefer 'deep' functions**: Avoid extracting very short code fragments (1-2 lines) into a separate function, especially if this function would require many arguments. Such shallow functions with wide interfaces increase complexity without meaningfully reducing line count. Instead, strive for *deep functions* (spanning multiple lines) with *narrow interfaces* (e.g., only 1-3 input arguments, i.e., **fewer arguments than the function has lines of code**), which tend to be more general and reusable [15].
> - **Address nesting**: If your code becomes overly nested, this can be a sign that parts of the code should be moved into a separate function. This simplifies logic and shortens lines.

- **Use Guard Clauses**: Deeply nested `if`-statements can make code harder to read. Instead, use guard clauses [1] to handle preconditions (e.g., checking for wrong user input) early, leaving the "happy path" clear and concise. For example:

```
if condition:
    if not other_condition:
        # do something
        return result
else:
    return None
```

Can be refactored into:

```
if not condition:
    return None
if other_condition:
    return None
# do something
return result
```

This approach reduces nesting and improves readability.

5.2 Documentation & Comments: A Note to Your Future Self

While you write it, everything seems obvious. However, when revisiting your code a few months later (e.g., to try a different experiment), you're often left wondering what the heck you were doing. This is especially true when some external constraint (like a library quirk) forced you to create a workaround instead of opting for the straightforward solution. When returning to such code, you might be tempted to replace the awkward implementation with something more elegant, only to rediscover why you chose that approach in the first place. This is where comments can save you some trouble. And they are even more important when collaborating with others who need to understand your code.

We distinguish between documentation and comments: **Documentation** provides the general description of ***when and how to use your code***, such as function docstrings explaining what the function computes, its input parameters, and return values. This is particularly important for open source libraries where you can't personally explain the code's purpose and usage to others. **Comments** help developers understand ***why your code was written in a certain way***, like explaining that unintuitive workaround. Additionally, for scientific code, you may also need to document the origin of certain values or equations by referencing the corresponding paper in the comments.

⚠ Code should be self-documenting

Ideally, your code should be written so clearly that it's self-explanatory. Comments shouldn't explain *what* the code does, only *why* it does that (when not obvious). **Comments and docu-**

mentation, like code, need to be maintained—if you modify code, update the corresponding comments, or they become misleading and harmful rather than helpful. Using comments sparingly minimizes the risk of confusing, outdated comments.
Informative variable and function names are essential for self-explanatory code. When you're tempted to write a comment that summarizes what the following block of code does (e.g., `# preprocess data`), consider **moving these lines into a separate function with an informative name**, especially if they contain significant, reusable logic.

Naming is hard

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
> – Phil Karlton[2]

Finding informative names for variables, functions, and classes can be challenging, but good names are crucial to make the code easier to understand for you and your collaborators.

Tips for effective naming

- Names should **reveal intent**. Longer names (consisting of multiple words in `snake_case` or `camelCase`, depending on the conventions of your chosen programming language) are usually better. However, **stick to domain conventions**—if everyone understands `X` and `y` as feature matrix and target vector, use these despite common advice denouncing single letter names.
- **Be consistent**: similar names should indicate similar things.
- **Avoid reserved keywords** (i.e., words your code editor colors differently, like Python's `input` function).
- Use **verbs for functions, nouns for classes**.
- Use **affirmative phrases for booleans** (e.g., `is_visible` instead of `is_invisible`).
- Use **plurals for collections** (e.g., `cats` instead of `list_of_cats`).
- **Avoid encoding types in names** (e.g., `color_dict`), since if you decide to change the data type later, you either need to rename the variable everywhere or the name is now misleading.

5.3 Tests: Protect What You Love

We all want our code to be correct. During development, we often verify this manually by running the code with example inputs to check if the output matches our expectations. While this approach helps ensure correctness initially, it becomes cumbersome to recreate these test cases later when the code needs changes. The simple solution? **Package your manual tests into a reusable test suite** that you can run anytime to check your code for errors.

Tests typically use `assert` statements to confirm that the actual output matches the expected output. For example:

[2] https://martinfowler.com/bliki/TwoHardThings.html

```python
def add(x, y):
    return x + y

def test_add():
    # verify correctness with examples, including edge cases
    # syntax: assert (expression that should evaluate to True), "error message"
    assert add(2, 2) == 4, "2 + 2 should equal 4"
    assert add(5, -6) == -1, "5 - 6 should equal -1"
    assert add(-2, 10.6) == 8.6, "-2 + 10.6 should equal 8.6"
    assert add(0, 0) == 0, "0 + 0 should equal 0"
```

Testing in Python with `pytest`

Consider using the `pytest` framework for your Python tests. Organize all your test scripts in a dedicated `tests/` folder to keep them separate from the main source code.

Pure functions—those without side effects like reading or writing external files—are especially **easy to test** because you can directly supply the necessary inputs. Placing your main logic into pure functions therefore simplifies testing the critical parts of your code. For impure functions, such as those interacting with databases or APIs, you can use techniques like **mocking** to simulate external dependencies.

Call-By-Value vs. Call-By-Reference

Depending on your programming language, function arguments can be passed either by **value (a copy of the variable's content)** or by **reference (the function accesses the variable's memory location, allowing direct manipulation)**. For clean code, we aim to follow the principle: "What happens inside a function stays inside the function (except for return values)." However, passing by reference can introduce unintended side effects, where changes made inside the function affect variables outside it, increasing complexity and leading to confusing bugs. In Python, this often occurs with mutable data types like lists and dictionaries:

```python
def change_list(a_list):
    a_list[0] = 42

if __name__ == '__main__':
    my_list = [1, 2, 3]
    print(my_list)  # [1, 2, 3]
    change_list(a_list)
    print(my_list)  # [42, 2, 3]
```

To prevent such side effects, create a copy of the variable before modifying it. This ensures the original remains unchanged:

```python
from copy import deepcopy

def change_list(a_list):
    a_list = deepcopy(a_list)
    a_list[0] = 42

if __name__ == '__main__':
    my_list = [1, 2, 3]
    change_list(a_list)
    print(my_list)  # [1, 2, 3]
```

To avoid sneaky bugs, **include tests to verify**:

1. **Inputs remain unchanged** after execution.
2. The function **produces the same output when called twice** with identical inputs.

When designing your tests, focus on **edge cases**—unusual or extreme scenarios like values outside the normal range or invalid inputs (e.g., dividing by zero or passing an empty list). The more thorough your tests, the more confident you can be in your code. Each time you make significant changes, run all your tests to ensure the code still behaves as expected.

Some developers even adopt **Test-Driven Development (TDD)**, where they write tests before the actual code. The process begins with writing tests that fail, then creating the code to make them pass. TDD can be highly motivating as it provides clear goals, but it requires discipline and may not always be practical in the early stages of development when function definitions are still evolving.

ℹ Testing at different levels

Ideally, you'll test your software at all levels:

- **Unit Tests:** Test individual components (e.g., single functions) to verify basic logic.
- **Integration/System Tests:** Check that different parts of the system work together as expected. These often require more complex setups, like running multiple services at the same time.
- **Manual Testing:** Identify unexpected behavior or overlooked edge cases. **Whenever a bug is found, create an automated test to reproduce it and prevent regression.**
- **User Testing:** Evaluate the user interface (UI) with real users to ensure clarity and usability. UX designers often perform these tests using design mockups before coding begins.

Debugging

When your code doesn't work as intended, you'll need to debug—**systematically identify and fix the problem**. Debugging becomes easier if your code is organized into small, testable functions covered by unit tests. These tests often help narrow down the source of the issue. If none of your tests caught the bug, write a new test to reproduce it and ensure this case is covered in the future.

To isolate the exact line causing the error:

- Use `print` statements to log variable values at key points and understand the program's flow.
- Add `assert` statements to verify intermediate results.
- Use a **debugger**, often integrated into your IDE, to set breakpoints where execution will pause, allowing you to step through the program manually and inspect variables.

Debugging is an essential skill that not only fixes bugs but also improves your understanding of the code and its behavior.

5.4 Make It Fast

> *Make it run, make it right, make it fast.*
> – Kent Beck (or rather this dad, Douglas Kent Beck[3])

Now that your code works and produces the right results (as you've dutifully confirmed with thorough testing), it's time to think about performance.

> **!** Readability over performance
>
> **Always prioritize writing code that's easy to understand.** Performance optimizations should never come at the cost of readability. More time is spent by humans reading and maintaining code than machines executing it.

Find and fix the bottlenecks

Instead of randomly trying to speed up everything, focus on the parts of your code that are actually slow. A quick way to find bottlenecks is to manually interrupt your code during a long run; if it always stops in the same place, that's likely the issue. For a more systematic approach, use a **profiler**. Profilers analyze your code and show you how much time each part takes, helping you decide where to focus your efforts.

Accessing files on disk or fetching data over the network is one of the slowest operations in most programs. Whenever possible, **cache the results** by storing the loaded data in memory to avoid repeated access to external resources. Just be mindful of how frequently the external data changes and invalidate the cache when the information becomes outdated.

> Run it in the cloud
>
> Working with large datasets may trigger `Out of Memory` errors as your computer runs out of RAM. While optimizing your code can help, sometimes the quickest solution is to run it on a larger machine in the cloud. Platforms like AWS, Google Cloud, Azure, or your institution's own compute cluster make this cost-effective and accessible. That said, always look for simple performance improvements first!

[3] https://x.com/KentBeck/status/704385198301904896

Think About Big O

Some computations have unavoidable limits. For example, finding the maximum value in an unsorted list requires checking every item—there is no way around this. The "Big O" notation is used to describe these limits, helping you understand how your code scales as data grows (both in terms of execution time and required memory).

- **Constant time** ($\mathcal{O}(1)$): Independent of dataset size (e.g., looking up a key in a dictionary).
- **Linear time** ($\mathcal{O}(n)$): Grows proportionally to data size (e.g., finding the maximum in a list).
- **Problematic growth** (e.g., $\mathcal{O}(n^3)$ or $\mathcal{O}(2^n)$): Polynomial or exponential scaling can make algorithms impractical for large datasets.

When developing a novel algorithm, you should examine its scaling behavior both theoretically (e.g., using proofs) and empirically (e.g., timing it on datasets of different sizes). Designing a more efficient algorithm is a major achievement in computational research!

Divide & Conquer

If your code is too slow or your dataset too large, try **splitting the work into smaller, independent chunks and combining the results**. Such a "divide and conquer" approach is used in many algorithms, like the merge sort algorithm, and in big data frameworks like MapReduce.

Example: MapReduce

MapReduce [4] was one of the first frameworks developed to work with 'big data' that does not fit on a single computer anymore. The data is split into chunks and distributed across multiple machines, where each chunk is processed in parallel (*map* step), and then the results are combined into the final output (*reduce* step).

For instance, if you're training a machine learning model on a very large dataset, you could train separate models on subsets of the data and then aggregate their predictions (e.g., by averaging them), thereby creating an ensemble model.

Replace For-Loops with Map/Filter/Reduce

Sequential `for` loops can often be replaced with `map`, `filter`, and `reduce` operations for better readability and potential parallelism:

- `map`: Transform each element in a sequence.
- `filter`: Keep elements that meet a condition.
- `reduce`: Aggregate elements recursively (e.g., summing values).

For example:

```python
from functools import reduce

### Simplify this loop:
result_sum = 0
result_max = -float('inf')
for i in range(10000):
    new_i = i**0.5
    # the modulo operator x % y gives the remainder when diving x by y
    # i.e., we're checking for even numbers, where the rest is == 0
    if (round(new_i) % 2) == 0:
        result_sum += new_i
        result_max = max(result_max, new_i)

### Using map/filter/reduce:
# map(function to apply, list of elements)
new_i_all = map(lambda x: x**0.5, range(10000))
# filter(function that returns true or false, list of elements)
new_i_filtered = filter(lambda x: (round(x) % 2) == 0, new_i_all)
# reduce(function to combine current result with next element, list of elements, initial value)
result_sum = reduce(lambda acc, x: acc + x, new_i_filtered, 0)
result_max = reduce(lambda acc, x: max(acc, x), new_i_filtered, -float('inf'))
# (of course, for these simple cases you could just use sum() and max() on the list directly)
```

In Python, list comprehensions also offer concise alternatives:

```python
new_i_filtered = [i**0.5 for i in range(10000) if (round(i**0.5) % 2) == 0]
```

Exploit Parallelism

Many scientific computations are "embarrassingly parallelizable," meaning tasks can run independently. For example, running simulations with different model configurations, initial conditions, or random seeds. Each of these experiments can be submitted as a separate job and run in parallel on a compute cluster. By identifying parts of your code that can be parallelized, you can save time and make full use of available resources.

5.5 Refactoring: Make Change Easy

Refactoring is the process of **modifying existing code without altering its external behavior** [8]. In other words, it preserves the "contract" (interface) between your code and its users while improving its internal structure.
Common refactoring tasks include:

- **Renaming**: Giving (internally used) variables, functions, or classes more meaningful and descriptive names.

- **Extracting Functions**: Breaking large functions into smaller, more focused ones (→ one function should do one thing).
- **Eliminating Duplication**: Consolidating repeated code into reusable functions.
- **Simplifying Logic**: Reducing deeply nested code structures or introducing guard clauses for clarity.
- **Reorganizing Code**: Grouping related functions or classes into appropriate files or modules.

Why refactor?

Refactoring is typically done for two main reasons:

1. **Addressing Technical Debt**:
   When code is written quickly—often to meet deadlines—it may include shortcuts that make future changes harder. This accumulation of compromises is called "technical debt." Refactoring cleans up this debt, improving code quality and making the code easier to understand.
   - Example: Revisiting old code can be like tidying up a messy campsite. Just as a good scout leaves the campground cleaner than they found it, a responsible developer leaves the codebase better for the next person (or themselves in the future).
2. **Making Change Easier**:
   Sometimes, implementing a new feature in your existing code feels like forcing a square peg into a round hole. Instead of struggling with awkward workarounds, you should first refactor your code to align with the new requirements. The goal of software design isn't to predict every possible future change (which is impossible) but to adapt gracefully when those changes arise. This promotes an evolutionary architecture, where you solve problems once you understand them better [6].
   - Before adding a new feature, clean up your code so that the change feels natural and seamless. This not only simplifies the task at hand but also results in a more general, reusable functions and classes.

Refactorings to simplify changes

> *For each desired change, make the change easy (warning: this may be hard), then make the easy change.*
> – Kent Beck[4]

- **Replace Magic Numbers with Constants**: Magic numbers—values with unclear meaning—can make code harder to understand and maintain. By replacing them with constants, you create a single source of truth that's easy to modify.
  ```
  # Before:
  if status == 404:
      ...

  # After:
  ```

[4]https://x.com/KentBeck/status/250733358307500032

```
ERROR_NOT_FOUND = 404
if status == ERROR_NOT_FOUND:
    ...
```

- **Don't Repeat Yourself (DRY)**: Copying and pasting code may seem like a quick fix, but it leads to problems later. If the logic changes, you'll need to update it everywhere it's duplicated, which is error-prone. Instead, move the logic into a reusable function or method.

```
# Before:
if (model.a > 5) and (model.b == 3) and (model.c < 8):
    ...

# After:
class MyModel:
    def is_ready(self):
        return (self.a > 5) and (self.b == 3) and (self.c < 8)

if model.is_ready():
    ...
```

- **Implement Wrappers**: When working with external libraries or APIs, their provided interface might not align with your needs, and adapting to it directly can lead to awkward implementations in your code. A better solution is to create a wrapper that implements **the interface you wish you had**, translating the external API's inputs and outputs into the format that best suits your implementation. This approach keeps your code clean, consistent, and easier to maintain, while confining the less-than-ideal API interactions to a single location. Plus, if the external API changes, you only need to update the wrapper instead of changing your code everywhere.
- **Use Alternative Constructors**: Similar to a wrapper, you can add a class method to create objects in a way that's different from the regular constructor. This is useful when the input data doesn't directly match what the constructor needs. For instance, imagine you have a configuration file that specifies settings for a simulation. If the names or structure of these settings don't match the constructor's parameters, you can create a `from_config` method to handle the translation and then call the constructor with the correct arguments. The advantage is that if the format of the configuration file changes in the future, you only need to update this one method, keeping the rest of your code the same.

```
class Date:
  def __init__(self, year, month, day):
      self.year = year
      self.month = month
      self.day = day

  @classmethod
  def from_str(cls, date_str):
      # Parse the string and create a new instance
      year, month, day = map(int, date_str.split('-'))
      return cls(year, month, day)
```

```
# Usage:
date1 = Date(2025, 01, 30)
date2 = Date.from_str("2025-01-30")
```

- **Organize for Coherence**: Keep code elements that need to change together in the same file or module. Conversely, separate unrelated parts of your code to prevent unnecessary entanglement. This way, changes are localized, which reduces cognitive load.

  In larger codebases shared by multiple teams, this is even more critical. **When changes require excessive communication and coordination, it signals a need to reorganize the code.** Clear ownership and reduced dependencies help teams work independently while keeping the system coherent through agreed upon interfaces.

Additional tips

- **Test as you refactor**: Always run tests before and after refactoring to ensure no functionality is accidentally broken. Writing or expanding automated tests is often part of the process to safeguard against regressions.
- **Leverage IDE support**: Modern IDEs like PyCharm or Visual Studio Code provide tools for automated refactoring, such as renaming, extracting functions, or moving files. These can save time and reduce errors.
- **Avoid over-refactoring**: While cleaning up code is valuable, avoid making unnecessary changes that don't improve functionality or clarity. Over-refactoring wastes time and can confuse collaborators.

By refactoring regularly and following these practices, you'll create a cleaner, more maintainable codebase that is adaptable to future needs and enjoyable to work with.

Before you continue

At this point, you should have a clear understanding of:

- How to transform your ideas into code.
- Some best practices to write code that is easy to understand and maintain.

6 From Research to Production

Your results look great, the paper is written, the conference talk is over—now you're done, right?! Well, in academia, you might be. But let's explore some concepts and tools that are common in industry and could take your code to the next level. Maybe they even inspire you to turn your project into a deployable app—an excellent reference when you apply for your next job!

6.1 Components of Software Products

So far, your code might consist of scripts or notebooks for analysis and a set of reusable helper functions in your personal library. The next step? Making your code accessible to others by turning it into standalone software with a graphical user interface (GUI). Furthermore, we'll explore how to expand beyond static data sources like CSV or Excel files.

Graphical User Interface (GUI)

Software shines when users can interact with it easily. Instead of using a command-line interface (CLI), these days, users expect intuitive GUIs with buttons and visual elements.

We can broadly categorize software programs into:

1. **Stand-alone desktop or mobile applications**, which users download and install on their devices.
2. **Web-based applications** that run in a browser, like Google Docs. These are increasingly popular thanks to widespread internet access.

For web apps, the GUI users interact with is also referred to as the **frontend**, while the **backend** handles behind-the-scenes tasks like data storage and processing. Even seemingly standalone desktop clients often connect to a backend server for cloud storage or to enable collaboration on shared documents. We'll explore how this works in the section on APIs.

In research, the goal is often to make results more accessible, for example, by transforming a static report into an interactive dashboard where users can explore data. To do this, we recommend you start with a web-based app.

Many books and tutorials were written on the topic of building user-friendly software applications and a lot of it is very specific to the programming language you're using—please consult your favorite search engine to discover more resources on this topic.

> Web apps with Python's `streamlit` framework
>
> If you use Python, try the Streamlit framework to create web apps from your analysis scripts in minutes.

Databases

So far, we've assumed that your data is stored in spreadsheets (like CSV or Excel files) on your computer. While this works for smaller datasets and simple workflows, it becomes less practical as your data grows or is generated dynamically, such as through user interactions, and needs to be accessed and updated by multiple people at the same time. This is where databases come in, offering a more efficient and scalable way to store, retrieve, and manage data [11].

Databases come in many forms, each suited to different types of data and use cases. Two key considerations when choosing a database are [17]:

1. the kind of data you need to store, and
2. how that data will be used.

Types of Data in Databases

Different kinds of databases are ideal for different types of data (see also Chapter 2):

- **Structured data:** This resembles spreadsheet data, with rows for records and columns for attributes. Structured data is typically stored in relational (SQL) databases, where data is organized into multiple interrelated tables. Each table has a schema—a strict definition of the fields it contains and their types, such as text or numbers. If data doesn't match the schema, it's rejected.

  > Normalization in relational databases
  >
  > A process called normalization reduces redundancy by splitting data into separate tables. For example, instead of storing `material_type`, `material_supplier`, and `material_quality` directly in a table of `samples`, you'd create a `materials` table with unique IDs for each material, then reference the `material_id` in the `samples` table. This avoids duplication and makes updates easier but requires more complex queries to combine tables and extract all the data needed for analysis.

- **Semi-structured data:** JSON or XML documents contain data in flexible key-value pairs and are often stored in NoSQL databases. Unlike SQL databases, these databases don't enforce a strict schema, which makes them ideal for handling complex, nested data structures or dynamically changing datasets. For example, APIs often exchange data in JSON format, which can be stored as-is to avoid breaking it into tables and reconstructing it later.
  Modern relational databases, such as Postgres, blur the line between structured and semi-structured data by supporting JSON columns alongside traditional tables.

- **Unstructured data:** Files like images, videos, and text documents are typically stored on disk. Data lakes (e.g., AWS S3) provide additional tools to manage these files, but fundamentally, this is similar to organizing files in folders on your computer. If your data is processed through a pipeline, it's a good idea to save copies of the files at each stage (e.g., in `raw` and `cleaned` folders).
- **Streaming data:** High-volume, real-time data (e.g., IoT sensor logs) is best managed in specialized databases optimized for streaming, such as Apache Kafka.

Use Cases for Databases

When choosing a database, you'll also want to consider how the data will be used later:

- **Transactional processing (OLTP):** In this use case, individual records are frequently created and retrieved (e.g., financial transactions). These systems prioritize fast write speeds and maintaining an up-to-date view of the data.
- **Batch analytics (OLAP):** Data analysis is often performed in large batches to generate reports or insights, such as identifying which products users purchased. To avoid overloading the operational database with complex queries, data is typically copied from transactional systems to analytical systems (e.g., data warehouses) using an ETL process (Extract-Transform-Load).
- **Real-time analytics:** For applications requiring live data (e.g., interactive dashboards), databases and frameworks optimized for streaming or in-memory processing (e.g., Apache Flink) are ideal.

Scaling your database system is another critical factor. Consider how many users will access it simultaneously, how much data they'll retrieve, and how often it will be updated.

CRUD Operations

Databases support four basic operations collectively called CRUD:

- **Create:** Add new records.
- **Read:** Retrieve data.
- **Update:** Modify existing records.
- **Delete:** Remove records.

These operations are performed using **queries**, often written in SQL (Structured Query Language). For example, `SELECT * FROM table;` retrieves all records from a table. If you're new to SQL, this tutorial is a great place to start.

ORMs and Database Migrations

Writing raw database queries can be tedious, especially when working with complex schemas. **Object-Relational Mappers** (ORMs) simplify this by mapping database tables to objects in your code. With ORMs, you can interact with your database using familiar programming constructs and even define the schema directly in your code.

When designing a database schema and implementing the corresponding ORM, it's helpful to first sketch out the structure of the data (Figure 6.1). Start by identifying the key objects (which map to database tables), their fields, and their relationships.

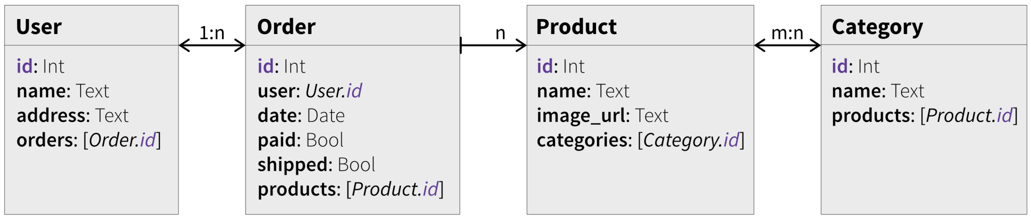


Figure 6.1: The classes `User`, `Order`, `Product`, and `Category` are mapped to the corresponding tables `users`, `orders`, `products`, and `categories`. Every record in a table is uniquely identified by a **primary key**, often named `id`. Fields in a table can either store data directly (e.g., text or boolean values) or reference records in another table, establishing **relationships between tables**. These relationships are defined using **foreign keys**, which store the primary key of a related record. For example, an `Order` references a single `User` ID (indicating the user who placed the order) and multiple `Product` IDs (the items included in the order).
Relationships between tables can be **bidirectional** or **unidirectional**, depending on the use case. For instance, when querying a `Product`, we want to list all the categories it belongs to, and vice versa. In contrast, the relationship between `Order` and `Product` only goes one way: when retrieving an `Order`, we want to know which products are included, but querying a `Product` doesn't usually require listing all the orders it appears in.

Database migrations, or schema changes, often require careful coordination between code and database updates. For instance, renaming a field means you have to update your database and modify the code accessing it at the same time. Keeping migration scripts and application code (including ORMs) in the same repository helps ensure consistency during such updates.

> **Managing databases in Python**
>
> The SQLModel library is highly recommended when working with relational databases in Python and also includes a great tutorial to learn more about ORMs and databases in general. For database migrations, check out Alembic.

APIs

In contrast to a user interface, through which a human interacts with a program, an **Application Programming Interface (API)** enables software components to communicate with one another. Think of it as a contract that defines how different systems interact. For example, an API might specify the classes and functions a library provides so developers can integrate it effectively.

APIs are often associated with **Web APIs**, which provide functionality over the internet. These can either be external services, like the Google Maps API for retrieving directions, or a custom-built **backend** that serves as an **abstraction layer for a database**. This abstraction is useful because it can combine data, enforce rules (e.g., verifying user permissions), and maintain a consistent interface even when the database structure changes.

Interacting with APIs

Web APIs typically use four HTTP methods that correspond to the CRUD (Create, Read, Update, Delete) operations in databases:

- **GET**: Retrieves data, most commonly used when accessing websites. You can include additional parameters by appending a `?` to the URL. For example, `https://www.google.com/search?q=web+api` searches for "web api" using the query parameter `q`. To pass multiple parameters, separate them with `&`.
- **POST**: Sends data to create a new record, often as a JSON object, for example, when submitting a form.
- **PUT**: Updates an existing record.
- **DELETE**: Removes a record.

You typically interact with APIs through a website's **frontend**, which triggers these API calls in the background. However, APIs can also be queried directly to access raw data, usually returned in **JSON** format.

> API Keys and Authentication
>
> Many APIs require an **API key** to access their functionality. This key serves as an identifier, allowing the API to authenticate users, track usage, and apply rate limits to prevent abuse. Always keep your API keys secure and avoid exposing them in public repositories or client-side code.

There are many free public APIs you can explore. As an example, we'll use The Cat API to demonstrate how to interact with an API.

You can perform a **GET request** directly in your web browser. For instance, by visiting `https://api.thecatapi.com/v1/images/search?limit=10`, you'll receive a JSON response containing a list of 10 random cat image URLs along with additional details like the image IDs.

For more advanced requests, such as **POST**, you'll need specialized tools. Popular GUI clients include Postman, Insomnia, and Bruno. If you prefer command-line tools, `curl` is a powerful option. Alternatively, you can interact with APIs programmatically using your preferred programming language and relevant libraries.

> Interacting with APIs programmatically
>
> In the examples below, we use `curl` and Python to interact with The Cat API to retrieve the latest votes for cat images with a GET request and submit a new vote using a POST request.
> **Using `curl`**
> Ensure `curl` is installed by running `which curl` in a terminal—this should return a valid path to your installation.

```
# GET request to view the last 10 votes for cat images
curl "https://api.thecatapi.com/v1/votes?limit=10&order=DESC" \
-H "x-api-key: DEMO-API-KEY"

# POST request to submit a new vote
# the payload after -d is the JSON object submitted to the API
curl -X POST "https://api.thecatapi.com/v1/votes" \
-H "Content-Type: application/json" \
-H "x-api-key: DEMO-API-KEY" \
-d '{
  "image_id": "HT902S6ra",
  "sub_id": "my-user-1234",
  "value": 1
}'
# now run the GET request again to see your new vote
```

Using Python

Python's `requests` library is great for working with APIs.

```python
import requests

BASE_URL = "https://api.thecatapi.com/v1/votes"
API_KEY = "DEMO-API-KEY"

# GET request to fetch the last 10 votes
def get_last_votes():
    response = requests.get(
        BASE_URL,
        headers={"x-api-key": API_KEY},
        params={"limit": 10, "order": "DESC"}
    )
    if response.status_code == 200:
        print(response.json())
    else:
        print(f"Error: {response.status_code}")

# POST request to submit a new vote
def submit_vote(image_id, sub_id, value):
    data = {"image_id": image_id, "sub_id": sub_id, "value": value}
    response = requests.post(
        BASE_URL,
        headers={"Content-Type": "application/json", "x-api-key": API_KEY},
        json=data
    )
    if response.status_code == 201:
        print("Vote submitted!")
    else:
        print(f"Error: {response.status_code}")

if __name__ == '__main__':
    get_last_votes()
    submit_vote("HT902S6ra", "my-user-1234", 1)
```

Implementing APIs

When designing an API, specifically a REST (REpresentational State Transfer) API, it's important to understand the concept of an **endpoint**. An endpoint is a specific URL in your API where a resource can be accessed or modified. For example, if you're building a photo-sharing app, an endpoint like `/images` might allow users to view or upload images. Endpoints should be named using descriptive, plural nouns (e.g., `/users`, `/images`) to clearly represent the resources being accessed. It's also best practice to avoid including verbs in endpoint names (e.g., `/get_users`), since the HTTP method (like GET or POST) already specifies the action being taken, such as retrieving or creating data.

Another key design principle is **statelessness**. Similar to the concept of pure functions, this means that each API request should contain all the information needed to complete the action, like user

authentication tokens. This way, the server doesn't need to remember anything about previous requests, making the API easier to scale. This is especially important in cloud-based environments where multiple requests from the same user may be routed to different servers [3].
Data that needs to be persisted can be stored either in the frontend or the backend database, depending on its purpose. Temporary data, like a shopping cart, can be maintained on the user's machine using cookies or local storage. Permanent data, such as a purchase order, is best stored in the backend database to ensure long-term accessibility. This approach supports stateless APIs, as the backend server doesn't need to keep the session state in memory. Instead, all necessary data is either included in the request or can be fetched from the database, allowing each request to be processed independently.

Implementing APIs in Python with `FastAPI`

FastAPI is a Python framework that makes building APIs straightforward. With just a few lines of code, you can turn functions into endpoints that validate input and return JSON responses. It's beginner-friendly and highly performant.

Asynchronous Communication

When your script calls a library function or API and waits for it to return before continuing with the rest of the code, this is an example of **synchronous communication**. It's similar to a conversation where one person speaks and then waits and listens while the other person responds.

In contrast, **asynchronous (async) communication** allows the program to **keep running while waiting for a response**. Once the response arrives, it is processed and integrated into the workflow, but until then the code just continues without it. Just like when you send an email to someone asking for some data and they send you the results a few hours later.
For example, a website might fetch data from multiple APIs, showing placeholders until the responses arrive. This approach improves the user experience because it keeps the user interface (UI) responsive and enables faster loading by **processing multiple tasks in parallel**.

Event-Driven Architecture

For most applications, communicating directly with external services—whether synchronously or async—is the right approach, because eventually the requested data is needed to finish the original task. But there are also use cases where it's enough that your message was received and you don't need to wait for a response. For example, when placing an order in an online shop, users only care that the order was submitted successfully. They don't wait in front of the screen until it was packaged and shipped—which could take days. An email notification can inform them of progress later.

Such a scenario calls for an **event-driven architecture**, which takes async communication to the extreme. Here, multiple services can operate independently by exchanging information via events using a **message queue (MQ)**, a system that temporarily stores event messages like JSON documents. These messages act as instructions, containing all relevant details about an event, such as a user's order information. Publishers (senders) create events, and subscribers (receivers) process them based on their type, such as `Order Submitted`.

An event-driven architecture offers several **advantages**. By decoupling components, it allows publishers and subscribers to run independently, even in different programming languages or environments. This makes it easier to scale systems and assign teams to own specific components without needing to understand the full system. Additionally, one event can trigger multiple actions. For instance, when an order is packed, one system might update the user database while another generates a shipping label. This approach thereby simplifies the propagation of data to multiple services and enables replication of live data to testing and staging environments.

However, this type of architecture also brings with it some **challenges**. Since no single component has a full view of the system, tracking the state of a specific task, such as whether an order is still waiting or in progress, can be difficult. Furthermore, while MQs often guarantee that each message is handled at least once, the system requires careful design in case a message is processed multiple times. For example, if a subscriber crashes after processing a message but before confirming its completion, the MQ might reassign the task to another instance, potentially leading to duplicate processing. For these reasons, event-driven architectures should only be used when direct communication between services is not an option [16].

Batch Jobs

Unlike continuously running services such as APIs, **batch jobs** are scripts or workflows used to process accumulated data in one go. They are particularly effective when tasks don't require immediate processing or when grouping tasks can improve efficiency. To automate recurring tasks, batch workflows can be scheduled at specific intervals using tools like **cron jobs**.[1]

Examples of scenarios where batch jobs are useful include:

- Fetching new messages from a queue every 10 minutes to process them in bulk, reducing overhead.
- Generating a sales report for the marketing department every Monday at midnight.
- Running nightly data validation to check for data drift or anomalous patterns.
- Retraining a machine learning model every week using newly collected data to create updated recommendations, like Spotify's "Discover Weekly" playlist.

For large-scale jobs, distributed systems might be necessary to ensure they complete within an acceptable timeframe.

Software Design Revisited

As your software evolves beyond a simple script, it becomes a system composed of multiple interconnected components. Each component can be viewed as a subsystem with its own defined boundaries and interface, responsible for specific functionalities while interacting with other parts of the system.

To manage this growing complexity, it's best to think of these components—such as a **GUI/frontend**, **API/backend**, and **database**—as distinct layers. A clean design follows the principle of **layered communication**, where each layer interacts only with the layer directly below it. For example, the frontend communicates with the backend, and the backend interacts with the database, avoiding "skip connections" where one layer bypasses another.

[1] Tools like Crontab Guru can help configure these schedules.

This design principle minimizes dependencies and makes the system easier to maintain: If the interface of one component changes, only the layer directly above it has to be adapted.

When you design these more complicated systems, it's even more important to sketch the overall architecture before you start with the implementation. Visualizing how the layers interact can reveal potential bottlenecks or unnecessary complexity and gives you and your collaborators clarity on the big picture.
Instead of lumping everything into "your code" versus "the outside world" as we did in Figure 4.3, you can use **swimlanes** to separate the process steps performed by each component to distinguish between the different layers (Figure 6.2).

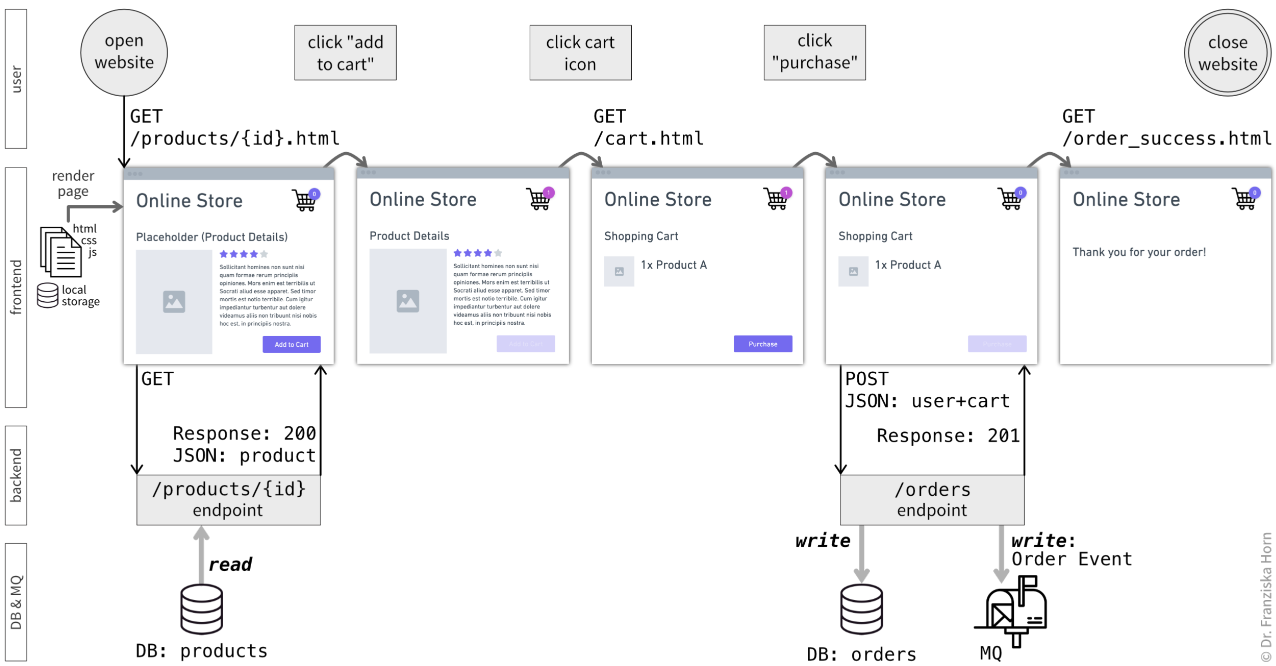


Figure 6.2: A simplified flow showing what happens when you order something online: A user opens a product page, which triggers a request to the API to fetch the corresponding product details. The user then clicks the "add to cart" button, which places the product into the shopping cart (in local storage managed by the frontend). The user then views the shopping cart and clicks "purchase", which triggers a POST request to the API, submitting the user's cart contents. The API creates a new record in the `orders` table to store the purchase details and submits an `Order` event to the Message Queue, thereby alerting other services that a new order needs to be packed and shipped. The endpoint returns with the status code 201 ("success") and the frontend redirects the user to a page that tells them the purchase was successful, at which point the user closes the tab.

6.2 Delivery & Deployment

Modern software development requires reliable and efficient processes to build, test, and deploy applications [7]. Delivery and deployment strategies ensure that new features and updates are released quickly, safely, and at scale, minimizing disruptions to users while maintaining quality.

CI/CD Pipelines: Automating Development Cycles

Continuous Integration (CI) and **Continuous Delivery/Deployment (CD)** pipelines are the backbone of modern software practices. CI focuses on automating the process of integrating code changes into a shared repository. Every change triggers automated tests to ensure that the new code works harmoniously with the existing codebase. CD extends this by automating the preparation or deployment of changes into production, either ready for manual approval (Continuous Delivery) or fully automated (Continuous Deployment). This drastically reduces manual effort, minimizes human error, and enables faster iteration cycles.

CI/CD pipelines are either included directly into version control platforms, such as GitHub Actions and GitLab CI/CD, or can be run using external tools like Jenkins or CircleCI.

Optimizing CI/CD Pipelines

To enhance pipeline efficiency and reliability, consider the following practices:

- **Dependency Caching:** Cache dependencies to reduce the time spent downloading and installing them for each build.
- **Selective Testing:** Run only the tests affected by recent changes to speed up feedback.
- **Real-Time Notifications:** Notify developers immediately when a pipeline fails, enabling faster issue resolution.

> **!** Security in CI/CD Pipelines
>
> Security must be a priority in any CI/CD process. For example, it is best practice to include a **dependency scanning** step to detect vulnerabilities in third-party libraries. Furthermore, you should never include **sensitive information**—such as API tokens, database credentials, or private keys—directly in your code. However, because CI jobs often require access to this information, you can securely store secrets using dedicated CI/CD variables or external **secret management tools** like HashiCorp Vault or AWS Secrets Manager.

A well-designed CI/CD pipeline not only saves time and resources but also ensures a consistent and high-quality delivery of software.

Containers in the Cloud

Containers, powered by tools like **Docker**, encapsulate applications with their dependencies, ensuring consistency across different environments. This portability simplifies deployment and reduces issues caused by environment differences.

For managing containerized applications at scale, **Kubernetes (k8s)** is the industry standard. Kubernetes automates the **orchestration of containers**, providing features like:

- **Auto-scaling:** Adjust resources dynamically based on workload.
- **Self-healing:** Automatically restart failed containers.
- **Load Balancing:** Distribute traffic efficiently across services.

Using Cloud Platforms

Cloud platforms like **AWS**, **Google Cloud Platform (GCP)**, and **Microsoft Azure** offer robust infrastructures for deploying and scaling applications. For simpler workflows, **managed services** like Render or Heroku abstract away much of the operational complexity.

Managing costs effectively is critical in cloud deployments. Key strategies include:

- **Resource Scaling:** Reduce unused resources during off-peak hours.
- **Serverless Computing:** Use serverless models, like AWS Lambda, for infrequent workloads to save costs.
- **Cost Monitoring Tools:** Leverage AWS Cost Explorer or GCP Billing to track and optimize spending.

Infrastructure as Code (IaC)

Instead of configuring your cloud setup manually through the platform's GUI, it is highly recommended to use **Infrastructure as Code** tools like Terraform and AWS CloudFormation to manage cloud infrastructure programmatically. The IaC configuration files can then be version-controlled, which ensures:

- Reproducible setups for consistent environments.
- Easier onboarding for new team members.
- Reduced risk of configuration drift.

Testing and Staging Environments

Deploying changes directly to production is risky. To ensure stability:

- Use **staging environments** that mimic production to validate changes before release.
- Maintain **testing environments** for early experimentation and debugging.

Techniques like **A/B testing** and **feature toggles** allow gradual rollouts or controlled exposure of new features, minimizing user disruption. This can be achieved using deployment strategies like:

- **Blue-Green Deployments:** Maintain two environments (blue and green) and switch traffic between them for A/B tests or to reduce downtime during updates.
- **Canary Releases:** Gradually expose updates to a small group of users, monitoring for issues before full deployment.

Scaling Considerations

As applications grow, scaling requires thoughtful architectural design. You should consider:

- **Task Separation:** For example, train machine learning models periodically as batch jobs, while keeping prediction services running continuously. This is particularly important when services have vastly **different user bases** (e.g., hundreds of admins versus millions of regular users), as they require varying replication rates for horizontal scaling. Especially if services rely on distinct dependencies, combining them into a single Docker container can result in a large, inefficient image, which increases the services' startup time.
- **Team Autonomy:** Design services so that teams can own and work on individual components independently, thereby reducing communication overhead and speeding up development cycles.

Monitoring and Observability

To ensure smooth operation and detect issues proactively, monitoring and observability are essential. Focus on:

- **System Performance:** Monitor the "golden signals"—latency, traffic, errors, and saturation. Tools like Prometheus and Grafana are commonly used.
- **Data Quality:** Track changes in input data distributions and monitor metrics like model accuracy to detect data drift.
- **Synthetic Monitoring:** Simulate user behavior to identify bottlenecks and improve responsiveness. Complement this with chaos engineering tools like **Chaos Monkey** to test your system's resilience by deliberately introducing failures, ensuring your infrastructure can handle unexpected disruptions effectively.
- **Distributed Tracing:** Debug across microservices using tools like Jaeger or OpenTelemetry.

When issues arise, having a rollback strategy is crucial. Options include:

- Reverting to a stable container image.
- Rolling back database migrations.
- Using feature toggles to disable problematic updates.

By combining robust delivery pipelines, thoughtful architecture, and effective monitoring, teams can ensure that their applications remain reliable, scalable, and adaptable to changing needs.

> Before you continue
>
> At this point, you should have a clear understanding of:
>
> - Which additional steps you could take to make your research project production-ready.

Afterword

You're still at the beginning of your journey towards professional software engineering. But I hope this book could give you a glimpse into what lies ahead.

I'm always looking to improve the contents of this book (or any other resources you can find on my website). Therefore, I would be eternally **grateful for your feedback**—whether you just found a typo, you think an explanation is unclear, or there are other topics that you think this book should cover—please send me an email to `hey@franziskahorn.de` to let me know what you think!

References

[1] Beck K. *Tidy First?* O'Reilly Media, Inc. (2023).
[2] Callaway E. Chemistry Nobel Goes to Developers of AlphaFold AI That Predicts Protein Structures. *Nature* 634(8034), 525–526 (2024).
[3] Davis C. *Cloud Native Patterns: Designing Change-Tolerant Software*. Simon; Schuster (2019).
[4] Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM* 51(1), 107–113 (2008).
[5] Foote B, Yoder J. Big Ball of Mud. *Pattern languages of program design* 4, 654–692 (1997).
[6] Ford N, Parsons R, Kua P, Sadalage P. *Building Evolutionary Architectures*. O'Reilly Media, Inc. (2022).
[7] Forsgren N, Humble J, Kim G. *Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations*. IT Revolution (2018).
[8] Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional (2018).
[9] Freiesleben T, Molnar C. *Supervised Machine Learning for Science: How to Stop Worrying and Love Your Black Box*. (2024).
[10] Horn F. *A Practitioner's Guide to Machine Learning*. (2021).
[11] Kleppmann M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc. (2017).
[12] Knaflic CN. *Storytelling with Data: A Data Visualization Guide for Business Professionals*. John Wiley & Sons (2015).
[13] McChrystal GS, Collins T, Silverman D, Fussell C. *Team of Teams: New Rules of Engagement for a Complex World*. Penguin (2015).
[14] Normand E. *Grokking Simplicity: Taming Complex Software with Functional Thinking*. Simon; Schuster (2021).
[15] Ousterhout JK. *A Philosophy of Software Design*. Yaknyam Press Palo Alto, CA, USA (2018).
[16] Richards M, Ford N. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Inc. (2020).
[17] Serra J. *Deciphering Data Architectures*. O'Reilly Media, Inc. (2024).