Practical Data Engineering for Geospatial Impact Evaluations

Seth Goodman[1,c] and Jacob Hall[1]

[1] AidData, The College of William & Mary, Williamsburg, Virginia, USA
[c] Corresponding author (smgoodman@wm.edu)

Table of Contents

Introduction

Data engineering is the design and implementation of systems, pipelines, and tools for working with data throughout its lifecycle. Familiar stages of a project where data engineering is involved include downloading and ingesting data from a third party source, cleaning and processing data for analysis, deploying code to run multiple variations of an analysis on a dataset, and exporting new analysis or data outputs for distribution or additional analytical work. While data engineering is a broad topic covering many different technologies, approaches, and practices, in this chapter we will narrow our focus to a subset of topics relevant to geospatial impact evaluations (GIEs). The goal is not to prepare you to become a data engineer, but to show you how *thinking like a data engineer* can improve your ability to effectively implement and scale GIEs. Along the way we will introduce core concepts, tools, and other information to facilitate your use of data engineering in GIEs, along with resources and examples you can build upon.

The use of geospatial data in impact evaluations has enabled researchers and practitioners to conduct cost effective evaluations of development interventions in cases where traditional approaches such as a preplanned RCT were not possible. The breadth of geospatial data coverage—both spatially and temporally—means that GIEs can be conducted at greater scales than would otherwise be possible, in addition to on projects that have already been completed. Despite the significant promise and benefits of GIEs, working with geospatial data at scale introduces cost and complexity to consider when preparing for an evaluation.

Regional time series coverage from modern satellite imagery products can easily result in terabytes of data for an evaluation and require significant computational capacity to process. While some datasets can be accessed in aggregate format from publicly available tools (e.g., GeoQuery) there is often a need for purpose-built processing of imagery to effectively evaluate project impacts. When using moderate resolution imagery products from Landsat 9 or Sentinel 2, or even sub meter products from Planet or Maxar, working with core surface reflectance data can become a computational task beyond the capabilities of basic desktop GIS software.

As the amount of processing and computational power required to prepare data for a GIE increases, the ease of reproduction decreases. As a result, developing clear and efficient workflows becomes critical to provide both a means of verifying the approaches used, and facilitating both reproduction as well as replication when possible. Continual improvements of geospatial data products means that the scope and complexity of geospatial data for impact evaluations will likely grow, and embracing the fundamentals of data engineering will be more important than ever.

To provide an effective overview of practical data engineering for GIEs, we have structured this chapter in three parts. Part 1 will first discuss the core concepts of how data engineering fits into a GIE. We will then introduce a set of tools you can get started with which we will employ throughout the chapter. This will set up your computational environment in a way that you can work through all the examples and exercises presented, and will serve as a foundation for future

work. Finally, we will provide a brief overview of the materials or core geospatial data types you will most often be working with and how they function.

In Part 2 we will dive into the practical application of data engineering for GIEs. The illustrative implementation will focus on downloading datasets, cleaning and processing data, and integrating data for analysis. Part 3 will take a step back to consider the barriers as well as potential solutions to common problems faced when working with geospatial data. We will also highlight relevant resources to facilitate your use of data engineering in future work, ranging from documentation for coding tools and data providers to additional literature on advanced concepts in data engineering.

Throughout the chapter, we will employ a set of tools and approaches for data engineering that are intended to provide an accessible and transferable experience. Our assumptions and choices are by no means the only way to utilize the concepts discussed, but hopefully presents the content in a way that will be easy for you to implement in a way that works for you. All code will be based on the Python programming language although many alternatives exist (e.g., R, Rust, C++, Java, Stata). Don't worry if you are not an expert in Python, as it typically has a pseudocode-like syntax that makes following along easy. Since Python is a very popular language, there is an abundance of support online for learning and troubleshooting.

Python has a particularly robust and active ecosystem of projects revolving around geospatial data as well as data engineering. While equivalent capabilities are possible in other languages, there may be fewer existing tools and projects for specific functionality. Later in this chapter we will explore parallelization and optimization approaches to address one of Python's weaknesses - speed - but ultimately leave it to you to decide between the lower development time and accessibility of Python and the speed of alternative, compiled languages. We will touch upon this topic again later when considering reproducibility and other data engineering concepts.

More broadly, we will focus on core concepts rather than specific or optimized implementations of specific functionality. For example, the introduction to data parallelization will emphasize approaching and preparing your data and processing pipeline in a way that can be split into pieces that can be run simultaneously rather than how to deploy specific infrastructure and software that can most effectively handle millions of computational tasks. We will briefly introduce notable options for tools as they are relevant to reflect different data engineering approaches or modalities (e.g., using a local desktop vs computing cluster vs cloud platform).

By the end of this chapter, you should be ready to approach your next GIE through a data engineering lens. Embracing the engineering mindset and focusing on developing code and data workflows that are reproducible and reusable will make conducting GIEs easier for you and your colleagues, and facilitate higher quality evaluations. As you progress through the chapter, we strongly encourage you to explore the examples and exercises, as well as engage with the recommended resources to get the full benefit. As valuable as the core concepts are, hands-on experience is often the best way to learn. You can find all code and related resources in the chapter GitHub repository available at: https://github.com/aiddata/data-engineering-for-gies.

Part 1: Foundations

★ Embracing engineering

The President of the National Academy of Engineering defined engineering as “the act of creating artifacts, processes, or systems that advance technology and address human needs using principles of the sciences, mathematics, computing, and operations.” [1] Historically, engineering fields such as mechanical, civil, or nuclear engineering have been underpinned by strong standards, accreditation, and accountability. Civil engineers designing a bridge are professionally licensed and liable for their work, which must follow national standards that detail aspects of the design from the quality control of steel member construction, to the installation of bolts.[2] While there are many differences between traditional engineering professions and new ones (software, data, and machine learning engineering to name a few) we will avoid a discussion on the use of the term engineering, and instead adopt the spirit and core facets of engineering, employing relevant practices to improve how we implement GIEs.

Even though data engineers and applications of data engineering do not have the same accreditation requirements or standards enforcement as civil engineering and similar engineering fields, we can strive to utilize similar principles to improve the quality and reliability of data workflows in GIEs, and make using data more efficient and effective. In this section we will introduce core concepts and best practices that will facilitate this aim.

Reproducibility and replication are essential tenets of all forms of scientific research that also apply to data engineering. Reproducibility refers to being able to run the same code with the same data and generate the same results. Replication refers to using the same code with new data to produce a similar finding. Both reproducibility and replication are highly relevant to GIEs. The ability to reproduce a GIE means that other researchers can run and validate the methods and findings from the evaluation. While replication is relevant to a GIE in terms of the findings associated with the intervention being studied, an even more appropriate concept for data engineering is reusability. Reusability refers to being able to take the core code and adapt it to a distinct GIE to reduce the amount of labor required. For example, code used to download and process NDVI data for a GIE in Ghana should be able to be adapted and reused with minimal effort to download and process NDVI data (from the same sensor) for Tanzania.

Reusability is more broadly a well established coding practice that underpins many programming paradigms such as functions, classes, and packages - all of which are designed to avoid having to generate duplicate code throughout your project. Although reusability is generally good to pursue, functions or classes can become increasingly complex in order to support multiple similar use cases. One school of thought is that an individual function should only perform a single task, and once something else is required, another function should be

[1] https://www.nae.edu/221278/Presidents-Perspective-What-Is-Engineering
[2] https://www.aisc.org/globalassets/aisc/publications/standards/a303-22w.pdf

created. Balancing the complexity of functions with reusability and readability (how easy code is to read, discussed more later) is a skill you will develop over time.

A related concept in data engineering is automation. Automation can be implemented at varying levels of complexity, ranging from entire workflows that are triggered and run without any human intervention (e.g. a dataset is updated on a provider database which is automatically downloaded and processed, and then a new analysis run) to a Python script which can be triggered manually and ensure all the workflow operations are run in the correct order. While the latter may be too simplified to qualify as automation for more experienced programmers, the reality is that many researchers and practitioners conducting GIEs do not have extensive programming backgrounds. It is common for those just starting out to have coding files - often interactive notebook style - with many effectively independent snippets of code that are each run manually. It is far too easy to end up with code and outputs that can not be reproduced when working that way. So even basic automation can go a long way to ensuring you and others can revisit your code and workflows in the future.

A common byproduct of creating reproducible and automated code is that your code base will typically be easier to test and conduct quality assurance (QA) on. For code that you intend to reuse often, and perhaps will become a dedicated package or utility, you may go as far as to implement unit tests. Unit tests are used to ensure individual functions or portions of the code are behaving as intended. This generally involves providing the function with a known input and verifying it produces the expected output. Unit tests can be extremely useful in catching unanticipated changes when modifying larger or more complex code bases. Pytest is a popular Python package for implementing unit tests and has a wide range of features.

For more limited code bases, an underutilized approach is to conduct code reviews with a colleague. Code reviews can take many forms, but ultimately just involve having someone else look over your code for errors. This can be as the code is written or after completion and on snippets of your code or on the finished product. In many instances, the act of explaining your code to another person can help to identify errors, and is also a valuable approach to debugging. If your code base doesn't warrant unit tests, and your team doesn't have the bandwidth for code reviews, an increasingly relevant option is using AI tools like ChatGPT. While any use of AI comes with a hefty warning to always double check an AI's output, it can be useful for identifying errors for you to investigate.

As you iterate through code - adding new features, debugging, conducting code reviews, etc - an essential development tool to incorporate is version control. Version control systems (VCS) are essentially a log of the changes made to your code every time you make a commit (a commit is a reference point to your code as a specific point in time, accompanied by a message detailing what you changed from the previous reference point). By using difference tracking - only keeping track of the changes made at each point in time, rather than saving full copies of the codebase at each point in time, you can efficiently review how you code has evolved and even quickly trace newly introduced bugs to the origin or roll back you code to an earlier bug-free version. One of the most valuable features of VCS is that it supports collaboration

across your team, and supports tools for merging code bases being worked on by multiple people. Git is one of the more commonly used and well known VCS and is often paired with the free cloud platform GitHub for managing code repositories and collaborations. We will explore using VCS with Git and GitHub more later.

Figure X: How git commit messages tend to degrade while working on a project. An example of what to avoid. https://xkcd.com/1296/

Documentation is another critical component of well designed data workflow (and use of code in general), that facilitates reproducibility and reusability. More often than not, you as the original developer of your code will be the one who benefits most from documentation. Eventually every programmer learns the price of hubris when they first think “I’ll remember why I wrote the code this way” only to revisit the code a year later and have no recollection of what they were thinking. There are many ways to create documentation that range from simple comments in your code to entirely standalone manuals or websites. The most important feature of documentation is to explain the reasoning behind code choices and how they provide functionality that would not otherwise be obvious. As we work through code later in the chapter, we will also provide examples of effective documentation.

Well documented code is a major component of sustainable development practices. In data engineering, and software development in general, sustainability refers to how well the code and application ages and whether it is difficult to maintain over time. For example, very complex and poorly written code with no documentation that depends on many obscure packages is likely to be very difficult to revisit and use in the future. Conversely, a focused and well written application with thorough documentation and reliance on well established packages will be much easier to update and maintain when necessary.

The ability to produce good, readable code is a skill developed over time and you should not be put off by a lack of experience. Even novice programmers can produce sustainable code by

leveraging established coding standards and tools to help enforce them known as linters. Linters can be run on your code, before you even run the code, to check for syntax errors, ensure you adhere to various coding conventions, and will help make your code more readable. These conventions help make your code approachable to others, who will be familiar with this standard style.

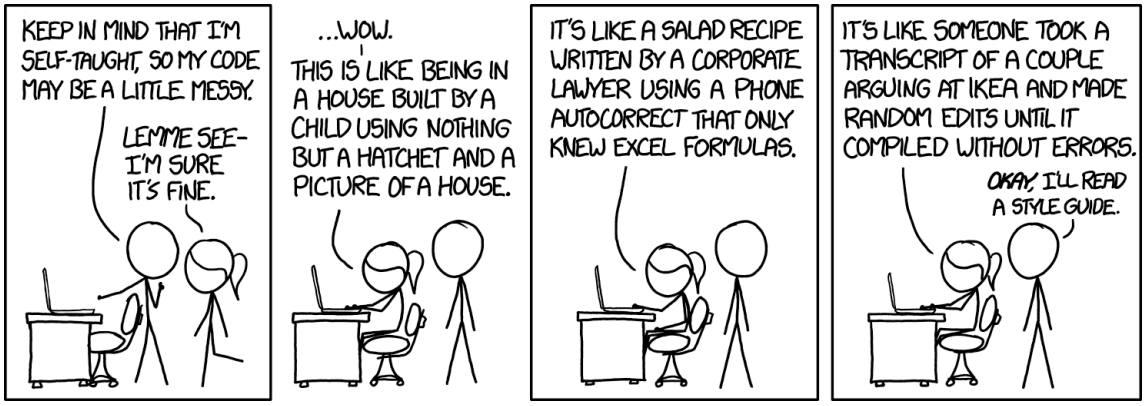

Figure X: Emphasizing the importance of coding style guides and standards. https://xkcd.com/1513/

External dependencies are another relevant component of sustainability we will touch upon further in terms of specific Python packages. Dependencies (referred to as packages in Python, or sometimes as libraries in other languages) are invaluable resources to avoid unnecessary work. There are many wonderful and active communities within the broader Python ecosystem that focus on developing and maintaining thousands of packages for various disciplines and specific functionality. Some of the criteria to consider when looking for packages include 1) whether the package is actively being developed/maintained, 2) how well established the package is, and 3) the quality of code/documentation. Inactive projects risk becoming unusable over time when new versions of the programming language come out but the package itself is not updated. However, for established packages with a very narrow focus, there is not always a need for ongoing development. In some instances, new projects with very rapid development may actually be more difficult to use as they may implement changes that are not backwards compatible while they refine the package. Later in the chapter we will introduce a few well developed Python packages that will serve most of your core data engineering needs for GIEs. The packages we will reference generally include well written and extensible code, good documentation, and active communities/development.

Leveraging code or data from third parties always involves some level of risk with regard to potential security and vulnerabilities. While these risks are generally low when working with reputable data providers and packages, it is always worth keeping in mind. Always download packages through an established packaging system (e.g., Pip or Conda for Python, discussed more below) and datasets from reliable sources (e.g., NASA, ESA, Harvard Dataverse). A more likely source of security concerns is how you store the passwords - often referred to as secrets or keys - used to access data repositories, APIs, or other platforms through your code. A good practice is to avoid storing secrets or keys in plaintext and never publish them to your code

repository.[3] Using local environment variables or SSH keys are approaches that enable you to still publish the relevant code, but keep your secrets and keys private.

There are many more best practices for programming and data engineering that you will come across over time as you become more advanced, but these core concepts will provide you with a solid foundation to start your data engineering journey. In the next section, we will move onto more applied topics as we work through setting up your environment and the tools you need.

★ Tools - Setting up your environment

The saying about choosing the right tool for the job is great as general advice, but when it comes to software, there are almost always many competing and viable options for any job. While your choice of tools is not inconsequential - and many people have strong opinions about their preferences - you do not want to get stuck making a decision rather than working on your actual project. In this section, we will present a set of tools in the form of software and approaches that are by no means the universally agreed upon “best” but are widely used, well established, and reliable. You may already have established preferences for alternatives or may develop preferences that differ from these along the way. Our goal is to provide you with a starting point and cover functionality that will translate to any comparable tool you may decide to use for an equivalent task.

One of the first considerations when setting up a coding environment is which code editor to use. There are many different options available, from full-featured integrated development environments (IDEs) to simple text editors. Some people may prefer editing code directly in a terminal, while others want a feature-rich graphical interface. Most popular options will support extensions for features like syntax highlighting and linting, which leverage language-specific information to highlight syntax in your code and help you to identify errors before you run the program. If you aren’t sure what editor is right for you, Visual Studio Code (VS Code) is a very popular choice that will work well out of the box, and supports all of the features and applications we will discuss in this chapter. If using VS Code to follow along in this chapter, we recommend installing the “Python” and “Pylance” extensions for a better Python experience.

[3] If you accidentally push commits to your code repository containing a secret or key, best practice is to reset your key or password.

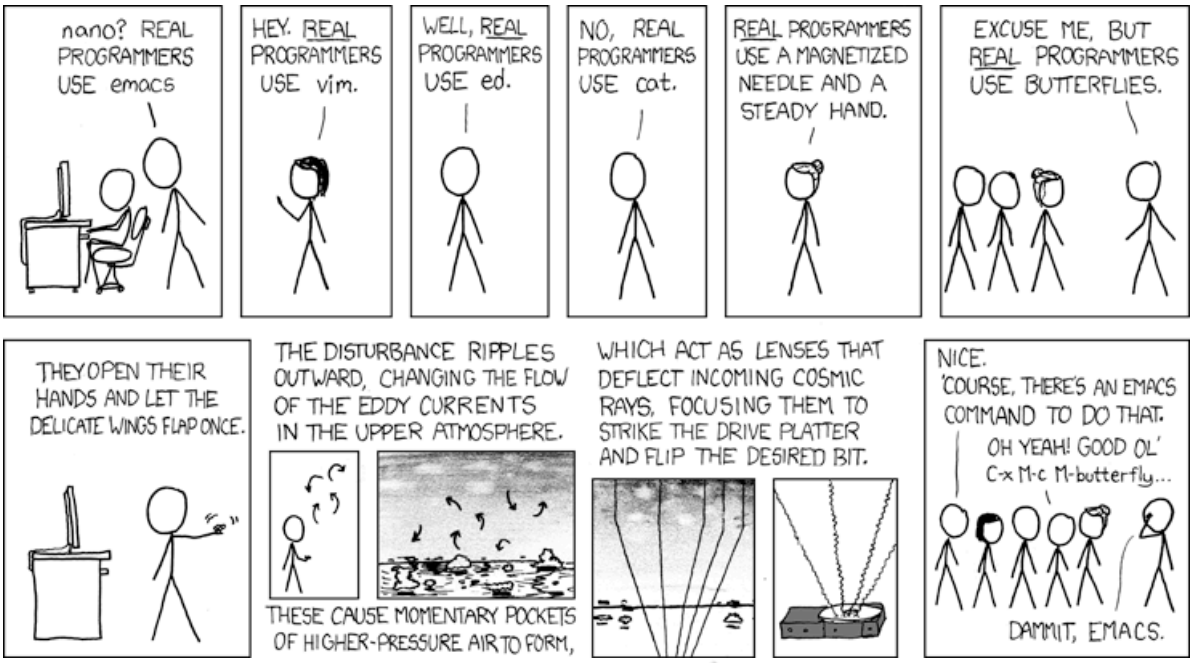

Figure X: A representation of the numerous opinions on what code editor is best. The correct answer is generally whatever works best for you. https://www.xkcd.com/378/

Another necessity for a coding project is a version control system (VCS) to allow you to track changes to your project over time, and share each version with others. By far the most popular system is git, which powers major code platforms like GitHub. Git is decentralized, meaning that anyone can keep an independent copy of a project, and merge them with each other at any time. A great feature of git is the “blame” functionality, which allows you to track changes made to a codebase and review who last edited specific lines of code as well as the associated commits. Git blame is not really intended to direct blame at others for issues, but is a tool to help you debug and understand the history of changes in a codebase.

GitHub is the most popular open code platform, and if you aren’t sure where (or if) to publish your code, we recommend starting there. Both git and GitHub have extensions available for VS Code to facilitate their use, though they are not required for using either. All code and related material for this chapter is available through a GitHub repository, which you can either download as a ZIP file (without a GitHub account or git) or by forking/cloning the account using a GitHub account to create your own copy of the repo.

> NOTE: Making your code public can be a great way to contribute to an expansive community of fellow researchers, programmers, and other interested people. When doing this, please consider choosing an open-source license for your work, so that others can confidently create derivatives of your work under well-defined terms. You may be surprised at the impact your work can have!

Depending on your computer's operating system, you may already have Python installed on your computer. However, we recommend against using that copy of Python for your analyses because it needs to remain intact to support various other applications. Instead, it is best to create project-specific environments (called virtual environments) that include their own copies of Python itself as well as any supporting packages. This has two primary advantages: 1) changes to one project will never affect the others, and 2) it will be easier for you and your collaborators to re-run code and reproduce outputs in the future by using the same stand-alone environment.

The Python community provides a package registry called PyPI where programmers can upload their work as packages for others to use. Virtually every Python package is available on PyPI, and installable using a Python package manager. While Python comes pre-installed with a package manager called pip, it is preferable to use a more holistic solution to package management that can also manage environments for you, as described above.

- As with pip, Python also comes preinstalled with a virtual environments system called venv. Using venv commands, you can create a new virtual environment and install packages into it. You can then "activate" your virtual environment and run Python code in it. This is a relatively simple, well-documented system.
- Another option, Conda, can build environments that not only include necessary Python packages but also lower-level libraries such as GDAL that are necessary for some geospatial analysis packages. The trade-off for this convenience is that Conda can sometimes take a long time to run, and environments are a bit trickier to share with others (since environments are usually built using platform-specific binaries). Overall, Conda is a tried-and-true solution with a large community. Miniconda is a version of Conda with fewer preinstalled packages that can be more efficient for some projects.
- There are a few newer package managers emerging for Python that promise to solve some of the pain points of incumbent options. At the time of writing, uv is gaining traction for its speed and simplicity. It offers a more streamlined workflow and replicates much of the core functionality of pip. However, it is worth noting that it's younger than Pip or Conda and could have rough edges, and may lack some advanced or use-case specific features.

For this chapter, we will use uv because it is convenient to set up, and has seen significant adoption in a short amount of time. Since it stores virtual environments in the standard venv format, it is compatible with other standard Python tools.

> NOTE: VS Code and other popular editors have native support for venvs, Conda environments, and others built right in. By connecting your editor to your coding environment, its syntax highlighting and terminal emulator will integrate directly with your environment and use the exact packages you have installed. See here for instructions on doing this in VS Code.

While there are a near endless number of useful Python packages available, we will highlight a few of the most relevant that will be used in this chapter. The exact set of packages used to replicate the exercises and examples in this chapter will be included in the GitHub repository.

- **numpy** is one of the most popular Python packages, providing array data types. It is implemented very efficiently, making it preferable over Python's built-in lists (or, lists of lists) when handling large amounts of data.
- **pandas** is another popular Python package, providing a DataFrame data type designed to store tabular data. Pandas provides many different functions for importing/exporting CSV, Parquet, and Microsoft Excel files (among many others) into DataFrames. Once represented as a DataFrame, it is relatively easy to clean and transform your data, and have it interact with other tools in the Python ecosystem.
- **shapely** provides geospatial primitives for vector data - such as Points, Lines, and Polygons - also well as many spatial operations for working with them (e.g., union, buffer, dissolve). Shapely is spatial but not geospatial, as it does not incorporate projection systems.
- **proj** provides an interface to the popular C++ library PROJ, which helps handle projections and coordinate reference systems. We will explore projections more in the following section.
- **geopandas** extends pandas, integrating shapely types and operations, as well as projections, into DataFrames. The GeoDataFrame type includes a "geometry" column, which stores the shapely type appropriate for your data. For example if you import a geospatial file containing administrative boundaries into a GeoDataFrame, the "geometry" column will likely contain Polygons (or MultiPolygons).
- **rasterio** is a useful tool for reading and writing raster files as numpy arrays, and includes a range of related functions for working with raster data.
- **fiona** is another Python package for reading and writing vector datasets. It supports Shapefiles and GeoPackages.

Much of the geospatial data ecosystem revolves around GDAL, a library that provides a wealth of functions to read and write geospatial data formats. Many of the Python packages listed above rely on GDAL for their underlying functionality. Depending on how you set up your environment, you may need to install GDAL onto your computer yourself. Conda usually finds a compatible version of GDAL automatically, but others will use whichever version of GDAL you install on your system.

It is also valuable to use desktop GIS software to graphically view and edit geospatial data. We highly recommend QGIS, which offers an extensive collection of tools for rendering, manipulating, and converting geospatial data. A company called Esri also sells a popular, proprietary alternative called ArcGIS. If you don't currently have GDAL installed, installing QGIS will also install GDAL for you.

> Note: Other Python packages may require specific system binaries such as GDAL to be installed in order to function properly or to enable certain features. Be sure to always read the documentation of a package and the installation requirements before using a new package.

While the above environment setup is a good starting point and will cover the examples and exercises for this chapter, you may find yourself working with more complex platforms/environments in the future. Examples of other environments include:

- A high performance computing (HPC) cluster at a university or research center
- A cloud platform such as those offered by Amazon, Microsoft, Google, and others
- A Kubernetes (k8s) or similar container based system
- Specialized cloud platforms such as Google Earth Engine (GEE)

Each of these alternative computing environments offer different benefits - largely focused on scaling workflows for larger datasets and more robust processes - but have drawbacks that typically come in the form of increased operational complexity and costs. We will explore these considerations more towards the end of the chapter when discussing potential barriers and solutions to data engineering workflows. A critical element of the concepts presented in this chapter is that they will fundamentally apply regardless of the environment you are working in. Some platforms may be designed to make certain elements easier to implement or abstract away other elements, but understanding core data engineering principles and approaches will aid you regardless of the specific tools you choose to use.

★ Materials - fundamentals of geospatial data

Understanding the materials you are working with is a critical component of any engineering discipline, whether that means steel beams, electrical components, or in our case, data. Geospatial data has come a long way since it was first transitioned from physical maps and surveys into digital GIS formats. Today, spatial features definitions, file formats, and numerous other specifications for related geospatial systems are largely standardized by the Open Geospatial Consortium (OGC)[4]. In a research context - where data is often collected, shared, and analyzed by different people - it's important to understand the nuances of how geospatial data is stored, organized, and accessed by both humans and computers in order to leverage the data and tools efficiently and effectively.

[4] See: https://www.ogc.org/standards/

Vector Data

There are two primary types of geospatial data: vector data and raster data. Vector data represents shapes as points in space, usually coordinates on a cartesian plane. “Points” and “lines” typically refer to XY coordinates (or, sets of coordinates) that represent geospatial features in a *coordinate reference system* (CRS). Vector data can consist of many *primitives*, or core feature types represented by different combinations of points. Below we detail the three most commonly used primitives:

- **Points**: Points are XY coordinates representing a single place. Points could be used to represent a fire hydrant, for example, or the location of a building.
- **LineStrings**: LineStrings (i.e., lines) are ordered lists of points that represent linear features such as roads or rivers.
- **Polygons**: Polygons are a closed ring of points representing an area. A polygon could represent an administrative unit or the footprint of a building.

In addition, the above feature types can exist as “Multi” versions. **MultiPolygons** are one of the more commonly used examples of this and are sets of Polygons, which can be used to represent holes or negative space(s) with a polygon. Some systems additionally include **MultiPoints** (sets of many Points), **MultiLineStrings** (sets of LineStrings), or **LinearRings** (closed rings of points that are not filled in like a Polygon would be).

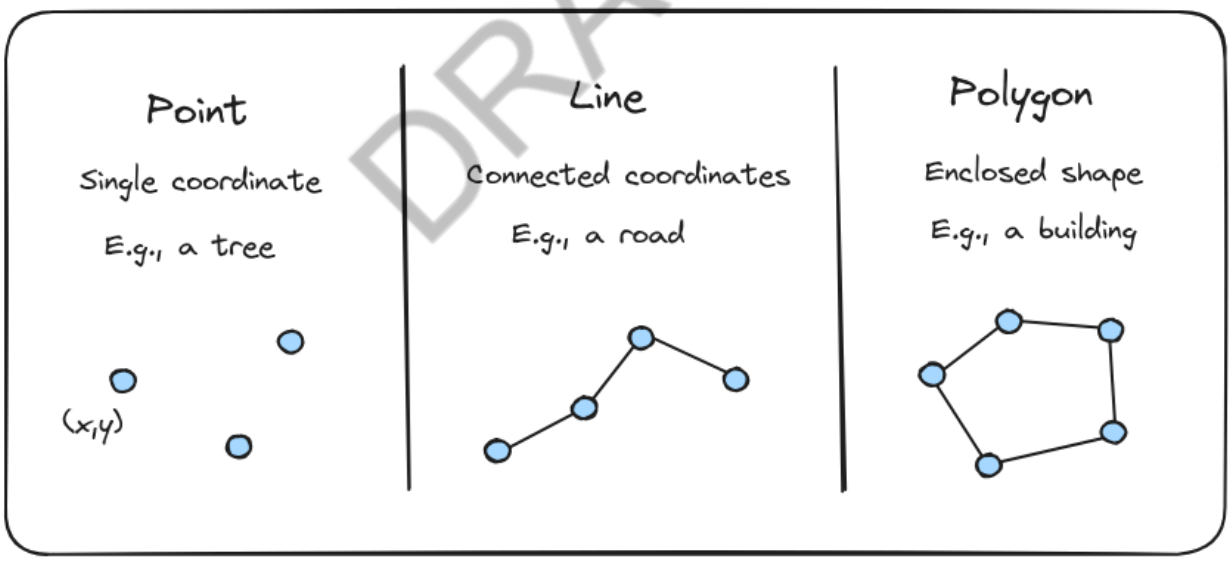

Figure X: Visualization of core vector feature types.

There are a variety of file formats for storing vector data. Among the most common are GeoJSONs, GeoPackages, and Shapefiles. GeoJSON is a particularly good choice for smaller datasets as it is simple to parse and supported by nearly every geospatial software. GeoPackages are another great choice, as they use the ubiquitous SQLite database format and therefore offer good query performance as the spatial features are indexed within the file format.

Shapefiles are an outdated, though still commonly used format. Depending on the specifics of an application, other formats such as GeoParquet, TopoJSON, or geospatial databases (e.g., PostGIS for PostGRES or DuckDB) may be worth exploring. We will primarily use GeoJSONs in this chapter for their simplicity, but GeoPackages are an excellent choice as you move on to more advanced workflows and larger datasets.

You may also see vector data stored using the Well Known Text (WKT) format. WKT is essentially vector data represented as a string, which can then be saved in any location such as a non spatial database or table. WKT is often seen in CSV files alongside other information, such as household survey data. Another common, simpler format for point data is recording the longitude and latitude in separate columns within a CSV.

Similar to storing spatial data alongside nonspatial data using WKT in a CSV, geospatial vector data formats can store additional data associated with each feature, known as attributes or properties. Attributes are key-value stores that can be used to record relevant information such as unique identifiers, descriptions, timestamps, and more. It's helpful to think of attributes as tabular data: if each spatial feature is a row, each attribute is the value of a column in that row.

Tabular data without an explicit geospatial component or a unique identifier for joining with vector data may instead include other fields that can be leveraged for joining. One example of such fields could be the names of each district in a country. In order to meaningfully compare this data with other geospatial information, it's necessary to *join* that data with the appropriate items from the geospatial dataset. To join this tabular data with geospatial vector data - i.e., the administrative boundary dataset for the districts (e.g. from geoBoundaries) - you would need to associated the district names from your tabular data with the district names from the vector data. Administrative unit names, and geospatial naming conventions in general, are notorious for a lack of standardization and may require some procedure to determine appropriate matches. This can be done manually, with a crosswalk table, or algorithmically by determining the similarity between names. Various other approaches may be used to link tabular and geospatial data including zip codes or other region-specific standards.

Raster Data

Raster data, in contrast to vector data, is represented by a grid of pixels stored like an image. Unlike a typical image taken by a camera, the grid of pixels is typically *georeferenced*: it includes metadata that describes its relation to the physical terrain of Earth. Each pixel contains one or more measured values (bands) for the area covered by that pixel. Satellites are a common source of raster data. When an image is taken of the Earth by a satellite, the extent of that image in terms of a coordinate reference system can be calculated in order to compare it with other sources of geospatial data. However, raster data can be created in many ways, including by rasterizing a vector dataset (converting from vector to raster) which we will discuss later in this section.

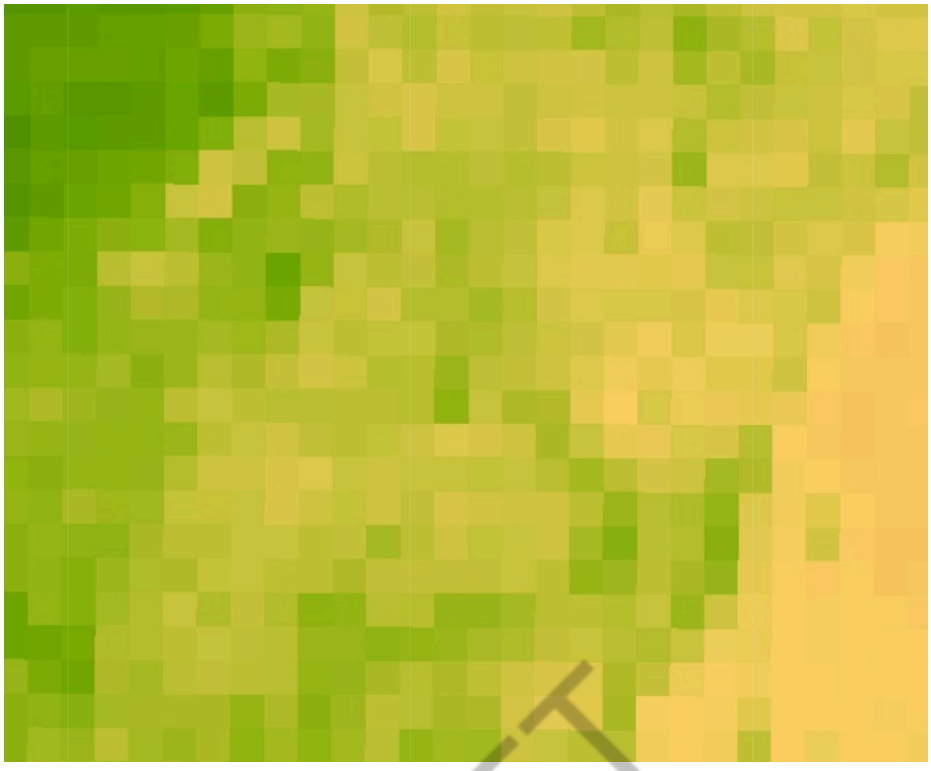

Figure X: Raster representing NDVI derived from Landsat imagery in Google Earth Engine. See: https://developers.google.com/earth-engine/datasets/catalog/LANDSAT_COMPOSITES_C02_T1_L2_ANNUAL_NDVI

Raster images are most commonly stored in the GeoTIFF file format - an open and georeferenced version of the common TIFF image file format. The Cloud Optimized GeoTIFF (COG) format is a newer variant of the GeoTIFF designed to make working with raster data over HTTP connections more efficient, allowing you to read only the parts of the file you need rather than downloading the entire file. Other file formats used for storing raster data included the Hierarchical Data Format (HDF) and NetCDF, along with many created for specific software or applications (e.g., proprietary ESRI formats, formats for map tiling for viewing in websites, formats specifically for DEM or LiDAR data).

Raster data will include one or more bands, or sets of values for each pixel that represent a single measurement. These bands could reflect the spectral range captured by the sensors recording the images (e.g., red, green, blue, infrared), time series data, or a single band with a distinct metric (e.g., population within each pixel). In the next section we will explore working with different bands of data in greater detail.

Data Transformations

A useful feature inherent to geospatial data is the ability to transform between vector and raster formats, and vice versa. Vector and raster transformations work because they are both

fundamentally ways of representing locations, just using different data structures. The most basic example would be how a single point is represented. As a vector a point consists of the longitude and latitude coordinate pair (X, Y). As a raster a point consists of a single pixel in a georeferenced image, where the pixel corresponds to the same longitude and latitude. While the point is defined by a precise location in the vector data, in the raster the precision is dependent on the resolution of the raster image's pixels. The pixel covering (X, Y) in a raster with 10 km resolution has a very different contextual meaning than a pixel covering the same (X, Y) at 1 meter resolution. As a result, when transforming between vector and raster data it is especially important to consider the resolution and the intended use of the resulting data. For example, a 10 km pixel vectorized to Point data could give the false impression that it is extremely precise data if the original pixel resolution was not known. Raster data can also be vectorized as Polygons instead of Points as we will see shortly.

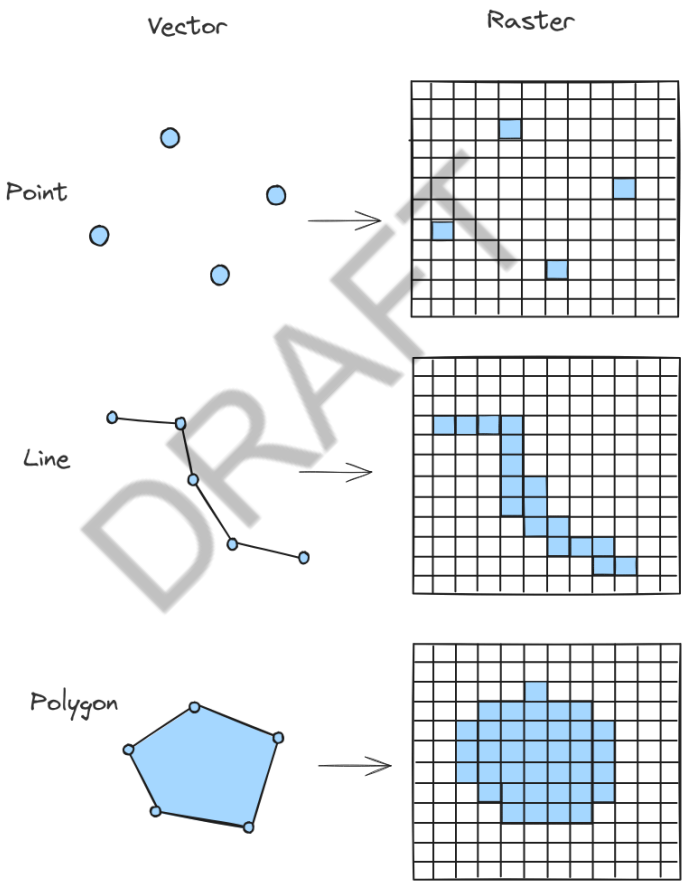

Figure X: Visualization of transforming vector data into raster format.

More complex feature types such as LineStrings and Polygons can be similarly transformed. Vector LineStrings would be rasterized as a series of connected pixels which the LineString spans, while a Polygon would be rasterized as a group of pixels along and within the boundary of the polygon. Instead of vectorizing raster data as points, as discussed earlier, we can also vectorize connected pixels of the same value in a raster e.g., all touching "1" pixels in a binary raster, or all touching "0" pixels) as in polygons. A common example of this type of vectorization

would be generating a raster of land cover classification estimates from satellite imagery, and then vectorizing each land cover class for subsequent analysis.

Coordinate reference systems

Coordinate reference systems (CRS) and projections are critical elements of any geospatial data - whether it is stored in vector, raster, other formats - that define how geospatial data is represented. Projection and CRS definitions are typically necessary to perform spatial joins, transformations, and other spatial operations. The CRS defines the specific framework within which spatial data is represented and the measurement / units used. For example, a CRS may represent latitude and longitude in degrees or in meters. Projections are used to define how a CRS and the associated geospatial data is mapped from the somewhat spherical (i.e., three dimensional) Earth onto a two dimensional surface. You may already be familiar with a few different map projections, like the (in)famous Mercator projection or perhaps an equal-area projection. When working with any geospatial data or latitude and longitude values, it is imperative to know what CRS is being used.

The most popular CRS in use today is WGS 84, which is the default for many file formats as well as large geospatial systems like GPS. However, **you should never assume** that data is WGS 84. There exist many, many other projections both similar to and radically different from WGS 84. For example, there is a set of projections called UTM that provide better spatial accuracy for a given region (“zone”). If you are analyzing data within a small area, you may benefit from *reprojecting* that data to an appropriate UTM zone for that area so that you can calculate more accurate distance measurements between points.

When it comes to working with and analyzing geospatial data, CRS are not only critical for visualizations but also for data integration (e.g., spatial joins, calculating intersections). When using two datasets with different CRS, most software will produce inaccurate results. While some software can handle multiple CRS dynamically (e.g., QGIS in many instances) you will often either get inaccurate results or simply an error message that you have different CRS.

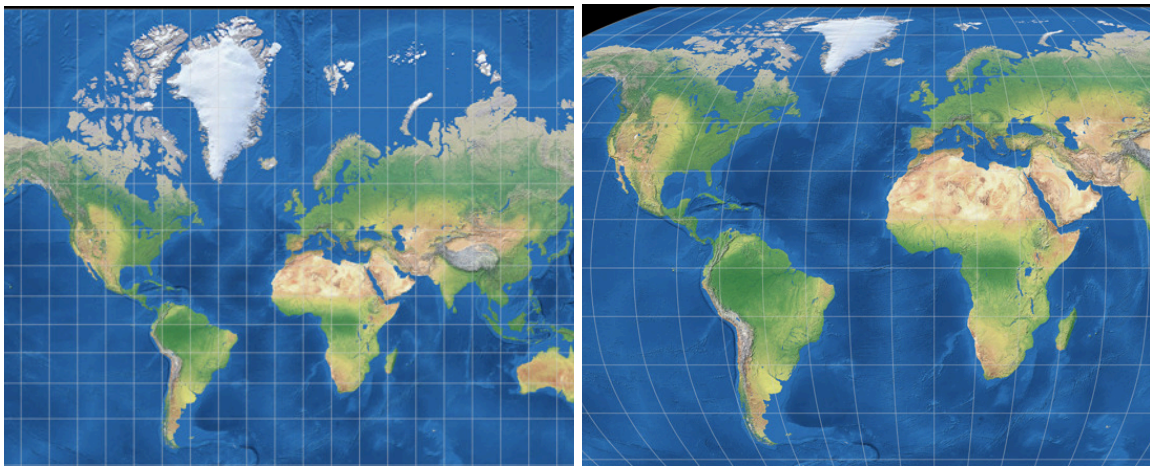

Figure X: Comparison of Mercator projection (left) and Robinson projection (right). Notice how in the Mercator projection Greenland appears similar in size to Africa, while the true land area of Greenland is more accurately reflected in the Robinson projection as a fraction of Africa's. Images by Tobias Jung CC BY-SA 4.0, via map-projections.net

The challenges presented by CRS and projections make Python packages like *proj* so important. When importing data, make sure to assign that data the correct CRS, and then reproject those values as necessary for your analysis. If you need to join any numeric coordinate values with each other, ensure that those coordinates are of the same CRS before proceeding.

When checking and transforming CRS or projections, always be certain you are actually transforming the data to the desired CRS rather than just redefining the CRS that is recorded for the data. If you change the recorded CRS without transforming the data, your data can become corrupted and may be tricky to recover (you would have to recall the original CRS, redefine it without transformation, then perform the intended transformation. If you shared the wrongly defined data without someone else who does not know the original projection, it would be nearly impossible for them to use it).

In the next section, we will employ the tools and data we introduced in this section to implement a practical data engineering workflow.

Part 2: Application

In this part of the chapter we will leverage the data engineering concepts, tools, and data types introduced to implement an illustrative data workflow. The goal will be to download and prepare boundary and raster data needed for a hypothetical GIE, to assess whether district level interventions focused on increasing farmer capacity led to a growth in cropland. We will first download administrative boundary vector data for the districts to represent the unit of analysis for the GIE, along with a time series (2015 and 2020) of land cover raster data. Next we will process the raster data to reflect the desired types of land cover relevant to the GIE. Finally we will extract land cover information from the raster data to the district boundaries, as well as join a record of hypothetical treatment data in tabular format to the geospatial data. The hypothetical treatment data will be reflective of a predetermined set of treated and comparison units as might be prepared for a quasi experimental research design used in a GIE.[5]

While this example will not be reflective of the workflow used in every GIE - and there are many potential elements of data engineering for GIEs we cannot fit into an introductory example - we aim to provide a starting point for translating the concepts previously introduced into actual code and workflows. We will mention additional topics that may be relevant to you as you create your own workflows, and later in the chapter we will dive into more specific technical barriers and solutions, as well as provide you with resources for further reading.

★ Setup

Before we dive into the actual dataset workflows, we will set up some important elements of the project. First, we will clone a copy of the project source code from the GitHub repository. One approach is to visit the repository at https://github.com/aiddata/data-engineering-for-gies, click the green “Code” dropdown menu, and then “Download ZIP.” Alternatively, you can fork the repository into your own GitHub account and connect to your new repository directly, allowing you pull down the current version to your environment and push commits as you adapt the code for your own application.

[5] Reference another chapter that deals more with GIE designs, and treatment/comparison groups.

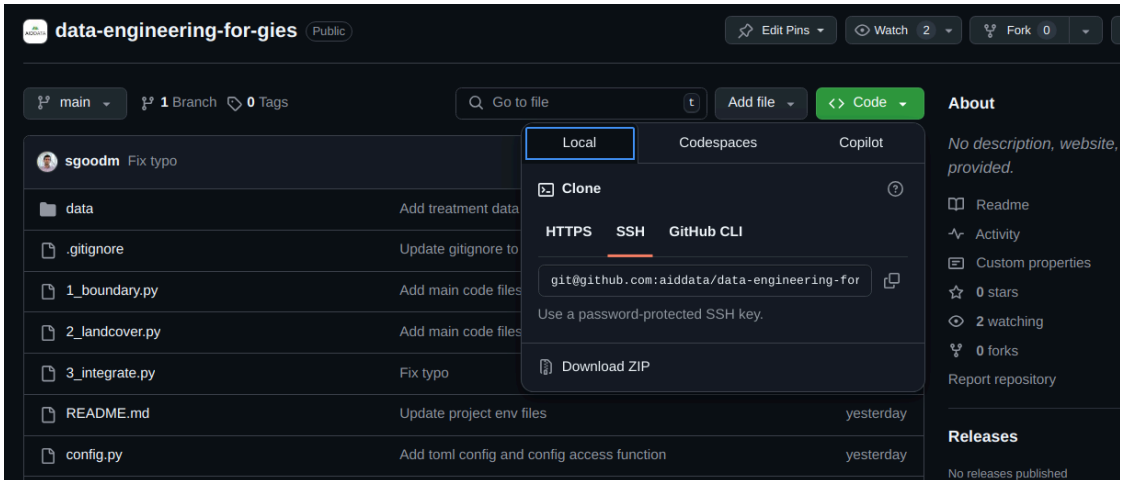

Figure X: The interface to download a GitHub repository.

Once you have the repository downloaded, we will create the uv virtual environment with the necessary packages. See the uv documentation for installation instructions specific to your operating system. With uv installed, open a terminal and cd into your project directory and use the uv sync command. Running this command for the first time you will see additional outputs in the terminal as the necessary packages are added to the environment.

```
~$ cd /home/userx/git/data-engineering-for-gies/
:~/git/data-engineering-for-gies$ uv sync
Resolved 62 packages in 0.72ms
Audited 56 packages in 0.01ms
```

Next, to control the parameters used in our workflow we will create a configuration file. A configuration file will allow us to change parameters in a single location to adjust how the workflow runs, rather than making changes throughout multiple code files which could result in duplication of parameter definitions and the potential for human errors. The configuration file we will use is based on a standardized yet flexible format known as TOML. There is a Python package available that makes reading TOML configuration files easy, and the format will suit a wide range of parameter definitions (numeric, string, list, nested items, etc.).

The default configuration file is provided in the repository within config.toml. We will include the full TOML file here and reference specific elements as we progress through the example. At the top level we define the working directory for our project, which you should update to the path of the downloaded repository in your local environment. No other parameters in the configuration should need to be updated to follow along with the rest of the example. We also include the path for a CSV file with the treatment definition that is relative to the base path (this is included in the repo). We then define two subsections of the configuration file using brackets for [boundary] parameters and [landcover] parameters. We will revisit these later as they are used.

```
base_path = "/home/userx/git/data-engineering-for-gies/data"
treatment_path = "treatment/ghana_adm2_treatment.csv"

[boundary]

version = "v6"
gb_data_hash = "9469f09"
gb_web_hash = "57dcd43"
dl_iso3_list = ["GHA"]
overwrite_existing = false

[landcover]

dataset_name = "esa_landcover"
api_key_env_var = "CDS_API_KEY"
overwrite_download = false
overwrite_processing = false
years = [2015, 2020]

mapping.0   = [0]
mapping.10  = [10, 11, 12]
mapping.20  = [20]
mapping.30  = [30, 40]
mapping.50  = [50, 60, 61, 62, 70, 71, 72, 80, 81, 82, 90, 100, 160, 170]
mapping.110 = [110, 130]
mapping.120 = [120, 121, 122]
mapping.140 = [140, 150, 151, 152, 153]
mapping.180 = [180]
mapping.190 = [190]
mapping.200 = [200, 201, 202]
mapping.210 = [210]
mapping.220 = [220]

category_map.no_data = 0
category_map.rainfed_cropland = 10
category_map.irrigated_cropland = 20
category_map.mosaic_cropland = 30
category_map.forest = 50
category_map.grassland = 110
category_map.shrubland = 120
category_map.sparse_vegetation = 140
category_map.wetland = 180
category_map.urban = 190
category_map.bare_areas = 200
category_map.water_bodies = 210
category_map.snow_ice = 220

```
We will implement a helper function to load the TOML data that can be used across all of the Python files in our workflow. The function takes a path, or defaults to “config.toml” if none is given, and checks that it exists. If the file does not exist it throws an exception, and if it does

exist, it loads the configurations using the tomllib package. The below code is saved to the config.py file.

```
from pathlib import Path
from typing import Union
import tomllib


def get_config(config_path: Union[Path, str] = "config.toml"
):
    config_path = Path(config_path)
    if config_path.exists():
        with open(config_path, "rb") as src:
            return tomllib.load(src)
    else:
        return FileNotFoundError("No TOML config file found for dataset.")
```

Lastly, as we proceed with the rest of the example you can either run code interactively within the environment using “uv run python” to enter the interpreter, or run the entire script directly using “uv run script.py” (where script.py is replaced with the relevant script). Given the implementation of the remainder of the code using classes, we recommend following along with the code in your editor but running the code in its entirety rather than in the interpreter.

★ Finding datasets

Finding the right data to facilitate the desired analysis is often the first major hurdle when implementing a GIE. Assuming the necessary data on outcome metrics or related covariates exist, but were not collected as part of your study or evaluation, you will often need to search through various data repositories and archives of geospatial data. Major data providers such as NASA and ESA are well indexed and searchable, and cover a wide range of scientific earth observation data from satellite sensors. However, there are numerous projects, research groups, and organizations responsible for maintaining valuable datasets that may not always be easy to find. While some data aggregators and hosting sites exist (e.g., Humanitarian Data Exchange, Google Earth Engine) you should expect to spend time performing a thorough search and literature review when looking for data on metrics you are not familiar with. The resources section towards the end of this chapter provides a number of different data repositories and lists of datasets that can provide a starting point when looking for geospatial data.

As you identify potential datasets for inclusions in your GIE, a critical step is evaluating the quality of the data. Reviewing dataset metadata, production methodology and related documentation, as well as understanding how the data provider itself distributes the data, are key components of evaluating a dataset. A lack of or difficult to find metadata and documentation is often an early warning sign when exploring datasets. If this critical information

about the quality of the dataset and how it was collected/produced does not exist, it is typically not reliable enough to use in a GIE. While metadata and documentation for high quality datasets should be reasonably easy to find, the reality is that most providers handle sharing this information differently and it may take some effort to find the right information even for datasets worth using in your GIE. Before deciding to include a dataset in your GIE, at minimum you should be able to determine how the dataset was produced (sensors or underlying data sources, processing methodology, etc), the spatial and temporal resolution, how often it is updated, and any caveats/limitations to using the data.

After you are confident in the dataset itself, the next consideration is regarding the data provider. One of the more frustrating aspects of developing data pipelines is when the data provider makes changes to the data or access methods that break your workflow. Breaking changes can take many forms and include altering the API or access criteria for the data, changing the file structure used to organize the data, create a new distinct version of the data (updating processing methodology, etc), changing who the actual provider of the data is, or in the worst case scenario, dropping support for the dataset entirely. While these are difficult to avoid entirely, you can take steps to mitigate their impact.

One significant consideration is the relationship between the data provider and the dataset itself. Data can often be hosted or replicated in multiple locations; often it is preferred to access the data directly from the original dataset producer, but that may not always be ideal. In most cases, accessing the data directly from the original producer will give you access to the current product and future updates without having to wait for a secondary provider to update the data in their own system. However, sometimes smaller dataset providers do not have the resources to host data themselves or their systems may be unreliable (i.e., research groups producing valuable data cannot always afford to develop an API and have a dedicated data distribution server). In some cases data producers may directly host their data in a third party repository, or you may determine that a third party repository is simply preferred for your workflow. Similarly, you may already have data pipelines for an established secondary data provider, such as GEE, which justify using it for additional datasets to make your broader collection of data pipelines more cohesive and maintainable.

Once you have selected a data source, you need to create a system for transferring that data from its origin onto your computer. To accomplish this, a number of questions must be answered:

- How is the data licensed? Is it legal to use the data for your intended purpose, and are there rules about how you distribute the data (or, that you may not do so)? If the license does not align with your intended use of the data, you may need to find a different dataset for your analysis (it never hurts to email the data provider to discuss this, as they may make exceptions for non-commercial / research applications).
- What protocol is used to host the data? Perhaps it is available via HTTP (i.e. as a link that could be downloaded in your web browser), or maybe it is hosted using FTP, or only using a custom API. Sometimes it is necessary to create web scraping systems to

download files that weren't hosted using an organized service. In rare cases, datasets may not have any programmatic approach for accessing them (though you can often get creative with web scraping and related tools), which should be a significant concern if you will need to rerun the data pipeline frequently (i.e., updating every month).
- Does the data source require authentication? If so, how are credentials obtained? Some data providers require that you make an account with them, or provision keys that expire after a period of time. If a provider manually approves data access requests, you may have to wait for a period of time before you get access.

It's critical to document your approach to solving these problems, both for your own future reference, and to promote replicability of your data pipeline by others. In the remainder of this section, we will work through the above and related steps of identifying and downloading the data for our illustrative GIE, with a focus on reliability and replication.

The first dataset we need is the administrative boundary information associated with the evaluation's unit of analysis. For our example, we will use district level or ADM2 boundary data and make two assumptions: 1) the hypothetical intervention aligns with the boundary data we will select, and 2) the administrative unit names of the geospatial boundaries align with the intervention treatment records. GIEs will often have existing and differing units of analysis defined for treatment locations that often do not align cleanly with other dataset in the event that they do not have a record of geospatial features themselves. A common scenario for our GIE example would be that the treatment record does not have geospatial data included and the boundary names do not perfectly match any existing geospatial boundary dataset. In these cases you would need to perform a fuzzy matching of the boundary names in the treatment and geospatial data, and if possible, engage with the original intervention implementers to validate the resulting matches. Units of analysis are rarely perfect at coarser levels such as ADM2 units, but it is always worth the effort to use the most accurate data possible.

Working through the data selection process described earlier, we will first consider potential sources of boundary data. Several popular options are geoBoundaries, GADM, and UN SALB. GADM has been used extensively historically, but the associated methodology and project activity is not well documented. SALB has a well documented methodology from a highly reputable source, but has limited coverage that could limit use for certain geographic areas. The geoBoundaries project is fully open source, well documented, and provides their source code and change history through GitHub (https://github.com/wmgeolab/geoBoundaries/). The project is also quite active with numerous contributors to the GitHub repository, and regular updates. Critically, geoBoundaries also has well defined API access for the data and does not require any special authentication. While we strongly encourage you to review these and other options in the context of specific GIE needs, geoBoundaries is often a good place to start when looking for boundary features.

There are also a number of potential land cover products available that could be used in this GIE. Popular land cover datasets include the ESA CCI Land Cover product at 350 meter resolution (1992-2021), the recently released 10 meter ESA World Cover product (2020 and

2021 only), MODIS land cover (2001-2022, 500 meter resolution), as well as numerous region specific products. Both the ESA CCI and MODIS land cover products are well established and include extensive documentation as well as convenient data access methods. While the land cover classification system varies slightly, it does not impede the use of either produce for this GIE. Given the general comparability of the two, we selected the ESA product due to the finer resolution. In the next section we will explore how to download both the boundary and raster data that we have selected.

★ Downloading Datasets

To produce modular and extensible code that will facilitate reproducibility and replication, we will create classes responsible for managing each dataset which contain methods for downloading and processing the data. We will work through the classes and their components in multiple stages covering first the downloading of both the boundary and raster data, and in the next section the processing. We will start with the boundary data (code found in 1_boundary.py in the repo) and move on to the land cover raster data (code found in 2_landcover.py).

To start building the boundary data class, we will first load the packages used, as well as the config helper function created earlier.

```
from pathlib import Path
from typing import List, Optional
import json
import logging

import shapely
import requests
import geopandas as gpd

from config import get_config
```

Next we will define a standalone function to retrieve data from a URL to facilitate accessing the geoBoundaries API. This uses the requests library to get the contents of a URL and then return the JSON representation of those contents.

```
def get_api_url(url: str):
    """
    Get the API URL and return the JSON object of content
    """
    response = requests.get(url)
    content = response.json()
    return content
```

We will then create and define the initialization function of the boundary dataset class. The __init__ function is called automatically when a new instance of a class is created. The arguments included in the init function should reflect the arguments passed in when a class instance is created. The arguments include the version of geoBoundaries to be used, the unique GitHub hash which reflects a specific instance of the geoBoundaries data history, and the unique GitHub hash associated with the geoBoundaries API version (the API hash actually dictates the data hash, but we kept the data hash as an additional sanity check to ensure the two agree). In addition, we provide the output directory for the downloads, whether to overwrite any existing downloaded data, and the list of country ISO3 codes for which to download data. The init function will then generate a version specific output path and create the version specific API link to be used for requests. Lastly, we will create a default dictionary for storing the metadata of each specific boundary file downloaded, which we will use and update later.

```
class geoBoundariesDataset():

    name = "geoBoundaries"

    def __init__(self,
                 version: str,
                 gb_data_hash: str,
                 gb_web_hash: str,
                 output_dir: str,
                 overwrite_existing: bool,
                 dl_iso3_list: Optional[List[str]] = None):


        self.output_dir = output_dir / f"{version}_{gb_data_hash}_{gb_web_hash}"

        self.overwrite_existing = overwrite_existing

        # leave blank / set to None to download all ISO3 boundaries
        self.dl_iso3_list = dl_iso3_list

        self.api_url =
f"https://raw.githubusercontent.com/wmgeolab/gbWeb/{gb_web_hash}/api/current/gbOpen/AL
L/ALL/index.json"

        self.default_meta = {
            "name": None,
            "path": None,
            "file_extension": ".geojson",
            "title": None,
            "description": None,
            "tags": ["geoboundaries", "administrative", "boundary"],
            "citation": "Runfola, D. et al. (2020) geoBoundaries: A global database of
political administrative boundaries. PLoS ONE 15(4): e0231866.
https://doi.org/10.1371/journal.pone.0231866",
            "source_name": "geoBoundaries",
            "source_url": "geoboundaries.org",
            "other": {},
            "group_name": None,
```

```
            "group_title": None,
            "group_class": None,
            "group_level": None,
        }
```

Next we will add a method that returns the logger for the class which we will use to output updates during the workflow. We will come back to using the logger later when running the script and viewing the outputs.

```
   def get_logger(self):
        """
        Retrieve and return the base logger to be used for this dataset
        """
        return logging.getLogger("boundary")
```

Next, we will define a method to determine which data from geoBoundaries to download based on the country ISO3 list provided in the configuration file. To do so, we retrieve the main manifest of all data available from the targeted version of geoBoundaries. We then iterate over the manifest to find all datasets which have the specified ISO3, and save the metadata provided by the API about each matched item. The metadata included a range of fields about the underlying source data for the boundary, descriptive statistics (vertice count, area, etc.) and critically the download URL for the specific version of the boundary dataset (the “gjDownloadURL” field). The full metadata dictionaries will be passed to the next method responsible for downloading each boundary file.


```
   def prepare(self):
        """
        Prepare data for download

        Retrieves the list of boundaries to download from the geoBoundaries API
        and filters it based on the provided ISO3 list.
        """
        logger = self.get_logger()

        logger.info(f"Preparing list of boundaries to download")

        api_data = get_api_url(self.api_url)

        if self.dl_iso3_list:
            ingest_items = [(i,) for i in api_data if i["boundaryISO"] in
self.dl_iso3_list]
        else:
            ingest_items = [(i,) for i in api_data]

        ingest_items = sorted(ingest_items, key=lambda d: d[0]['boundaryISO'])
        return ingest_items
```

```

The core method of the class is responsible for actually downloading each boundary dataset. After running the previous “prepare” function for our desired Ghana ISO3 (“GHA”) we have three boundary datasets to download corresponding to ADM0 (country), ADM1 (region), and ADM2 (district). The metadata dictionary for each of these datasets will be passed to the download function (dl_gb_item).

For each boundary dataset, the class will load the logger, save the specific ISO3 (as you could provide a list of different ISO3 in other use cases), and make a copy of the default metadata object created earlier when initializing the class. We then update the metadata object with the name and other fields specific to the dataset.

```
   def dl_gb_item(self, item: dict):
        """
        Download and process a single geoBoundaries item

        Prepared the metadata and downloads the boundary data from geoBoundaries.
        """
        logger = self.get_logger()

        iso3 = item["boundaryISO"]

        adm_meta = self.default_meta.copy()

        adm_meta["name"] = f"gB_v6_{iso3}_{item['boundaryType']}"

        logger.info(f"Processing geoBoundaries item: {adm_meta['name']}")

        adm_meta[
            "title"
        ] = f"geoBoundaries v6 - {item['boundaryName']} {item['boundaryType']}"
        adm_meta[
            "description"
        ] = f"This feature collection represents the {item['boundaryType']} level
boundaries for {item['boundaryName']} ({iso3}) from geoBoundaries v6."
        adm_meta["group_name"] = f"gb_v6_{iso3}"
        adm_meta["group_title"] = f"gB v6 - {iso3}"
        adm_meta["group_class"] = (
            "parent" if item["boundaryType"] == "ADM0" else "child"
)
        adm_meta["group_level"] = int(item["boundaryType"][3:])

        # save full metadata from geoboundaries api to the "other" field
        adm_meta["other"] = item.copy()
```

Using the boundary file download URL, we then generate the target output paths for the GeoJSON and metadata JSON, and ensure the associated parent directory exists. If the GeoJSON and JSON already exist and the overwrite option was not used, the boundary file will
```

be skipped and the function returns. If the files do not exist or overwrite was enabled, the download proceeds.

```
        # Example URL:
        #
"https://github.com/wmgeolab/geoBoundaries/raw/c0ed7b8/releaseData/gbOpen/AFG/ADM0/geo
Boundaries-AFG-ADM0.geojson",
        commit_dl_url = item["gjDownloadURL"]

        fname = Path(commit_dl_url).stem
        dir_path = self.output_dir / fname
        dir_path.mkdir(exist_ok=True, parents=True)

        geojson_path = dir_path / f"{fname}.geojson"
        adm_meta["path"] = str(gpkg_path)

        json_path = geojson_path.with_suffix(".meta.json")

        if geojson_path.exists() and json_path.exists() and not
self.overwrite_existing:
            logger.info(f"Skipping existing file: {geojson_path}")
            return
```

To download the data, we first attempt to read the boundary GeoJSON URL directly into a GeoDataFrame. While this often works, there are some cases where GeoPandas may not be able to read the data directly (e.g., file size limits for reading over URL). To overcome this, in cases where the direct read fails, we manually load the raw URL contents and then load those into a GeoDataFrame. If the second approach fails as well, potentially indicating the geoBoundaries website is unreachable, an error is logged.

```
        logger.debug(f"Downloading {commit_dl_url} boundary")
        try:
            gdf = gpd.read_file(commit_dl_url)
        except:
            if requests.get(commit_dl_url).status_code == 404:
                logger.error(f"404: {commit_dl_url}")
                return
            else:
                try:
                    raw_json = get_api_url(commit_dl_url)
                    gdf = gpd.GeoDataFrame.from_features(raw_json["features"])
                except:
                    logger.error(f"Failed to download {commit_dl_url}")
                    return
```

Once the data is downloaded we validate that the boundary contains a shapeName field. This is to address an edge case where some boundaries in geoBoundaries had incorrect schemas

(though this has likely been corrected since the specific version referenced in our example). Finally, we save the boundary data as a GeoJSON, and finalize the metadata by adding the boundary file extents and exporting the metadata JSON.

```
        if "shapeName" not in gdf.columns:
            potential_name_field = f'{item["boundaryType"]}_NAME'
            if potential_name_field in gdf.columns:
                gdf["shapeName"] = gdf[potential_name_field]
            else:
                gdf["shapeName"] = None

        gdf.to_file(geojson_path, driver="GeoJSON")

        logger.debug(f"Getting bounding box for {commit_dl_url}")
        spatial_extent = shapely.box(*gdf.total_bounds).wkt
        adm_meta["spatial_extent"] = spatial_extent


        # export metadata to json
        export_adm_meta = adm_meta.copy()
        with open(json_path, "w") as file:
            json.dump(export_adm_meta, file, indent=4)
```

The “main” method of the boundary dataset class controls the flow of class methods in order to generate the list of files to download and then run each download. The main method will be called after initialization of the class as we will explore next.

```
    def main(self):
        """
        Main function to run the geoBoundaries download process
        """
        logger = self.get_logger()

        # get the list of boundaries to download
        ingest_items = self.prepare()

        logger.info("Running boundary data download")

        # run the download tasks
        for item in ingest_items:
            logger.info(f"Processing {item[0]['boundaryISO']} boundary")
            self.dl_gb_item(item[0])

        logger.info("Finished downloading boundary data")
```

To actually initialize and run the class when the script is called, we will utilize a special block of code that checks if __name__ == "__main__". This block is only accessed if the script is run

directly. For example, if you wanted to use the boundary dataset class as part of another script, when you import the class in that file this block would not run.

Within this block we first load the config data by calling the get_config function. We will use the config to create a boundary specific config object containing the arguments required by the boundary dataset class. We will then set and make sure the boundary dataset output directory exists. In this section we will also initialize the logger by defining the output path for the log file, and setting the log level and logging format. Finally, we will initialize the class with the boundary config and call the main method to run all the code described above.

```
if __name__ == "__main__":

    config = get_config()
    boundary_config = config["boundary"]

    boundary_config["output_dir"] = Path(config["base_path"]) / "geoBoundaries"

    boundary_config["output_dir"].mkdir(parents=True, exist_ok=True)

    # set logging configuration
    logging.basicConfig(filename=boundary_config["output_dir"] / 'boundary.log',
                        filemode='a',
                        level=logging.INFO,
                        format='%(asctime)s - %(levelname)s - %(message)s')

    gBD = geoBoundariesDataset(**boundary_config)
    gBD.main()
```


Next we will start to implement the class to manage the land cover raster downloading and processing. While you can manually download the ESA CCI land cover products without an API key (using https://maps.elie.ucl.ac.be/CCI/viewer/download.php) we will create and account and generate an API key in order to build an data pipeline which can be automated.

First, register an account and log in to the Copernicus Climate Data Store (https://cds.climate.copernicus.eu/). Then go to your profile and the licenses page (https://cds.climate.copernicus.eu/profile?tab=licences) and agree to the following licenses:

- ESA CCI licence
- VITO licence
- Licence to use Copernicus Products

Now go back to your main profile page and scroll down to generate/copy your API key. Save the API key to a ".env" file in your project directory with the contents as displayed below.

```
```

```
CDS_API_KEY="YOUR-KEY-HERE”
```

Then in your terminal cd to the project directory and run the following to set the .env file as the environment file for uv. You will need to include this line of code in any automation scripts or run it in each new terminal before you run the code to download the land cover data.

```
export UV_ENV_FILE=.env
```

To start creating our land cover dataset class (found in 2_landcover.py) we will import the necessary packages, including the config helper used earlier.

```
import os
import shutil
import zipfile
from pathlib import Path
from typing import List
import logging

import cdsapi
import numpy as np
import rasterio

from config import get_config
```


Then we will create the class and init function. The init function will take in arguments for the raw download data directory, the temporary data processing directory, and the final output directory. It will also take a list of years (integers), your API key, overwrite flags for the download and processing stages, and a mapping dictionary which defines collections of land cover classes to group together in a single category for simplified classification (e.g., various types of forest classifications are all classified simply as “forest”). The dictionary defining this mapping is then vectorized, or turned into a function that can be used to look up the appropriate new classification given the original classification value (e.g., 61 would return 50). In addition, the init function will create a specific file using your API key which the Copernicus Data Store API checks for authentication when you submit requests. Finally the init function will establish a client connection to the Copernicus Data Store API.

```
class ESALandcover():

    name = "ESA Landcover"

    def __init__(self,
                 raw_dir: str,
                 process_dir: str,
                 output_dir: str,
```

```
                years: List[int],
                api_key: str,
                overwrite_download: bool,
                overwrite_processing: bool,
                mapping: dict):

        self.raw_dir = Path(raw_dir)
        self.process_dir = Path(process_dir)
        self.output_dir = Path(output_dir)
        self.years = years
        self.api_key = api_key
        self.overwrite_download = overwrite_download
        self.overwrite_processing = overwrite_processing

        self.v207_years = range(1992, 2016)
        self.v211_years = range(2016, 2022)

        cdsapi_path = Path.home() / ".cdsapirc"
        with open(cdsapi_path, "w") as f:
            f.write(
                f"url: https://cds.climate.copernicus.eu/api \nkey: {self.api_key}"
            )

        self.cdsapi_client = cdsapi.Client()

        vector_mapping = {int(vi): int(k) for k, v in mapping.items() for vi in v}
        self.map_func = np.vectorize(vector_mapping.get)
```

The mapping contained in the config TOML file which will be passed to the land cover class:

```
mapping.0   = [0]
mapping.10  = [10, 11, 12]
mapping.20  = [20]
mapping.30  = [30, 40]
mapping.50  = [50, 60, 61, 62, 70, 71, 72, 80, 81, 82, 90, 100, 160, 170]
mapping.110 = [110, 130]
mapping.120 = [120, 121, 122]
mapping.140 = [140, 150, 151, 152, 153]
mapping.180 = [180]
mapping.190 = [190]
mapping.200 = [200, 201, 202]
mapping.210 = [210]
mapping.220 = [220]
```

We will use a logging method as covered in the boundary class earlier, and then create the download method. The download method will take a year's integer value as the input and is responsible for retrieving the associated land cover raster.

The download method will first determine a version specific parameter based on the year specified (different product versions are used for different years). The output path for the

compressed data will be generated and we will check if it exists or if overwrite has been enabled. The download meta object needed for the API call is then created and the API call submitted.

```
    def download(self, year: int):
        """Download the ESA land cover data for a given year
        """
        logger = self.get_logger()

        if year in self.v207_years:
            version = "v2_0_7cds"
        elif year in self.v211_years:
            version = "v2_1_1"
        else:
            version = "v2_1_1"
            logger.warning(f"Assuming that {year} is v2_1_1")

        dl_path = self.raw_dir / "compressed" / f"{year}.zip"
        print(dl_path)

        if not dl_path.exists() or self.overwrite_download:
            dl_meta = {
                "variable": "all",
                "format": "zip",
                "version": [version],
                "year": [str(year)],
            }
            self.cdsapi_client.retrieve("satellite-land-cover", dl_meta, dl_path)
```


Once downloaded as a compressed ZIP file, we will unzip the data using Python's built-in zipfile package..

```
        zipfile_path = dl_path.as_posix()
        logger.info(f"Unzipping {zipfile_path}...")

        with zipfile.ZipFile(zipfile_path) as zf:
            netcdf_namelist = [i for i in zf.namelist() if i.endswith(".nc")]
            if len(netcdf_namelist) != 1:
                raise Exception(
                    f"Multiple or no ({len(netcdf_namelist)}) net cdf files found in
zip for {year}"
                )
            output_file_path = self.raw_dir / "uncompressed" / netcdf_namelist[0]
            if not os.path.isfile(output_file_path) or self.overwrite_download:
                zf.extract(netcdf_namelist[0], self.raw_dir / "uncompressed")
                logger.info(f"Unzip complete: {zipfile_path}...")
            else:
                logger.info(f"Unzip exists: {zipfile_path}...")

        return output_file_path
```

```

In the next section, we will continue to build out our dataset classes to perform the data processing steps, and then run both scripts.

## ★ Dataset processing

After successfully identifying, accessing, and downloading a dataset from a provider into your computational environment, the next step is processing the raw data to suit the needs of your GIE and facilitate data integration in subsequent steps. While we mentioned the importance of documentation and metadata availability earlier when discussing finding datasets, now is the time to leverage that information. Reviewing the metadata and potential quality assurance (QA) layers for the data products you are using will guide how to prepare and process the data. Your initial review of the metadata should have ensured that the dataset was fit for the intended use (e.g., sufficient resolution for measuring metrics relative to your unit of analysis) but now we build upon that to make sure our use of the data is accurately reflects the measurements provided by the data.

One common example of processing data is dealing with null or irrelevant values. While the null or NA data value is often defined in raster file formats and can is natively ignored by many geospatial data functions and tools, you should always 1) confirm the behavior of the packages/functions/tools you are using, and 2) confirm whether the NA value is set in the dataset.

In some cases, you may also wish to treat non NA values as if they were NA. For example, some vegetation indexes contain a continuous range of values for the index itself (e.g., 0 to 1), a NA data value (e.g., -9999), along with other discrete values representing water/ice/snow (e.g., -1, -2, -3). If you were to take the average value of this data within a unit of analysis - even if your function ignores the NA data - you would still be incorporating the negative values for water/ice/snow into your metric and distorting the anticipated vegetation index value. To deal with situations such as this, the data processing stage could set the additional water/ice/snow values to the NA data value. Then, the average value within a unit of analysis - ignoring the NA data - would only reflect the actual vegetation index measurements.

Similarly, multispectral imagery products are often provided with a separate layer (either a band in the imagery file or more commonly as a separate raster file) often known as the QA band. The QA band will contain various information about NA data, cloud coverage, and other pixel level metrics that can be used to appropriately process the data for your application. For example, one application may wish to set all pixels that are high confidence cloud cover to the NA value, while others may wish to set all high/medium/low confidence values to the NA value.
```

There are typically a range of QA band values to provide precise information about the type of cloud cover or other information for each pixel. [6]

For our application, we will be processing the raw land cover raster data in order to 1) simplify the land cover classification categories, and 2) convert the data to the Cloud Optimized GeoTIFF (COG) file format. Building on the code established in the previous section, we will implement a new function called “process” that will take two paths as arguments. The first is the path where we downloaded the raw data, and the second will be the desired output path. Within the function we will first set up the logger as done before, and then check whether the output path already exists and the status of the overwrite flags in the configuration file. If the overwrite was set for the download, but not the processing, the function will throw a warning as this would be an unusual scenario. If the output file exists, and the processing overwrite is not set, the function will simply skip to the end and avoid performing the processing again. If the file does not exist, or if it does and the processing overwrite is set, we move on to the portion of the function with the processing logic.

```
    def process(self, input_path: Path, output_path: Path):
        logger = self.get_logger()

        if self.overwrite_download and not self.overwrite_processing:
            logger.warning("Overwrite download set but not overwrite processing.")

        if output_path.exists() and not self.overwrite_processing:
            logger.info(f"Processed layer exists: {input_path}")

        else:
            logger.info(f"Processing: {input_path}")
```


The initial portion of the processing will prepare the file paths. First we will create a temporary processing directory. A temporary processing directory is not always needed, but can be very useful when working in computational environments with different file systems. For example, long term data storage is often based on drives from an entirely different computer (server) within the network. Depending on your internal network speed and traffic from other users/applications, constantly reading or writing data between your compute server and the storage server can be inefficient and potentially cause issues for others who are also using the storage server.

To avoid this, we copy the raw data to the storage on your local compute machine for processing, performing the processing and outputting another temporary file, then copying the temporary output file back to the storage server. For smaller datasets that can be read or written quickly, and typically would easily fit into memory on your compute machine this is generally not necessary. However, for larger datasets that do not fit into memory and require iterative reading

[6] As an example of pixel QA band, see the Landsat documentation: https://www.usgs.gov/landsat-missions/landsat-collection-2-quality-assessment-bands

and writing in smaller chunks - which results in significantly more I/O actions - this approach is usually more efficient.

After creating the temporary processing directory, we define the paths for the temporary copy of the raw input data and the temporary copy of the output data. We then copy the raw data from the storage server to the temporary local storage. Finally, we define the NetCDF specific path for the temporary input raster that can be used to read the data.

```
            self.process_dir.mkdir(parents=True, exist_ok=True)

            tmp_input_path = self.process_dir / Path(input_path).name
            tmp_output_path = self.process_dir / Path(output_path).name

            logger.info(f"Copying input to tmp {input_path} {tmp_input_path}")
            shutil.copyfile(input_path, tmp_input_path)

            logger.info(f"Running raster calc {tmp_input_path} {tmp_output_path}")
            netcdf_path = f"netcdf:{tmp_input_path}:lccs_class"
```

Next, we will implement the core processing logic. To do so, we first define meta information we wish to set for the output file. In this case, we are setting the output file format as a COG, and using LZW compression to reduce the file size. Other meta information fields can be added as defined by the rasterio package documentation. [7] A number of meta fields are required, but rather than redefine all of these, we can simply copy the existing meta from the input file and update it with our desired meta settings.

After opening the temporary NetCDF input file using rasterio and a context manager, we confirm that all block shapes within the raster data are the name. Block shapes are essentially tiles or chunks of the raster that are used to segment the data for storage and to facilitate reading and writing. The src.block_shapes is a list of all the block shapes, and the set function reduces this list to unique items. If the set length is 1, that means that all block shapes are the same. We then make a copy of the raw meta and update it with our meta.

Next, we also open the temporary output file, using the temporary output path and updated meta dictionary object. WIthin this context manager, we will then iterate over each block window or chunk of data. For each block window, we load the actual data associated with the window and then apply the mapping function defined earlier. As a recap, the mapping function is taking predefined sets of land cover classification values and assigning them to a single “parent” value. For example, the values [50, 60, 61, 62, 70, 71, 72, 80, 81, 82, 90, 100, 160, 170] all represent different types of forest classifications which we will simplify into a single classification represented by the value 50.

[7] See: https://rasterio.readthedocs.io/en/stable/topics/profiles.html

The last step of the processing is to set the datatype of the updated block data to ensure it matches the data type of the raster we set with the meta, and then write the block to the temporary output raster. The for loop will iterate through all the blocks and perform this read, process, and write until the entire raster has been completed. Both rasterio context managers will then automatically close when the code within them completes and the function returns.

```
            default_meta = {
                "driver": "COG",
                "compress": "LZW",
                # 'count': 1,
                # 'crs': {'init': 'epsg:4326'},
                # 'nodata': -9999,
            }

            with rasterio.open(netcdf_path) as src:
                assert len(set(src.block_shapes)) == 1
                meta = src.meta.copy()
                meta.update(**default_meta)
                with rasterio.open(tmp_output_path, "w", **meta) as dst:
                    for ji, window in src.block_windows(1):
                        in_data = src.read(window=window)
                        out_data = self.map_func(in_data)
                        out_data = out_data.astype(meta["dtype"])
                        dst.write(out_data, window=window)
```

Finally, we copy the temporary output file back to the storage server where it can be accessed in subsequent steps of the data workflow for the GIE.

```
            logger.info(f"Copying output tmp to final {tmp_output_path}
{output_path}")
            shutil.copyfile(tmp_output_path, output_path)
```

Similar to the boundary class, the main method will control the flow of logic for the download and processing methods of the land cover class when called. The method first ensures the necessary download directories exist and then iterates over each year specified to download the appropriate raster file. For each year downloaded, a list of the downloaded file path and the desired output path for the final product is created. This list is then iterated over and passed to the process function.

```
   def main(self):
        """
        Main function to run the ESA land cover data download and processing
        """
        logger = self.get_logger()

        os.makedirs(self.raw_dir / "compressed", exist_ok=True)
```

```
        os.makedirs(self.raw_dir / "uncompressed", exist_ok=True)

        # Download data
        logger.info("Running data download")

        download_results = []
        for year in self.years:
            download = self.download(year)
            download_results.append(download)


        os.makedirs(self.output_dir, exist_ok=True)

        # Process data
        logger.info("Running processing")

        process_inputs = zip(
            download_results,
            [self.output_dir / f"esa_lc_{year}.tif" for year in self.years],
        )

        for pi in process_inputs:
            _ = self.process(*pi)

        logging.info("Finished processing land cover data")
```

Again as in the boundary script, we use the special “main” code block to create the class instance and call the methods when the script is run. In this block we load the config file and create a landcover specific config object containing the necessary arguments for the land cover class. We then make sure the final output directory exists, set up the logger as we did earlier, create the land cover class, and call the main method.

```
if __name__ == "__main__":

    config = get_config()
    base_path = Path(config["base_path"])
    lc_config = {
        "raw_dir": base_path / config["landcover"]["dataset_name"] /"tmp/raw",
        "process_dir": base_path / config["landcover"]["dataset_name"] /
"tmp/processed",
        "output_dir": base_path / config["landcover"]["dataset_name"],
        "years": config["landcover"]["years"],
        "api_key": os.environ[config["landcover"]["api_key_env_var"]],
        "overwrite_download": config["landcover"]["overwrite_download"],
        "overwrite_processing": config["landcover"]["overwrite_processing"],
        "mapping": config["landcover"]["mapping"],
    }


    lc_config["output_dir"].mkdir(parents=True, exist_ok=True)
```

```
    # set logging configuration
    logging.basicConfig(filename=lc_config["output_dir"] / 'dataset.log',
                        filemode='a',
                        level=logging.INFO,
                        format='%(asctime)s - %(levelname)s - %(message)s')


    ESA_LC = ESALandcover(**lc_config)
    ESA_LC.main()
```

The methods will take some time to download and process the data, with the exact time varying based on your network speed and computing power. Once the script has finished processing you can explore the log and then open the output raster files to ensure there were no errors and familiarise yourself with the data visually.

In the next section we will take the boundary and raster data we downloaded and processed and explore how they can be integrated alongside other data for analysis in a GIE.

★ Data integration

Once you have downloaded and cleaned / processed the individual datasets you are ready to begin integrating the data for use in your GIE. While the specific format of your analysis ready data will depend on your GIE strategy, geospatial information commonly will need to be assessed after aggregating or extracting the data to the unit of observation. Typically, this stage of the analysis will leverage geospatial information but will not be inherently geospatial itself. Statistical models to facilitate analysis of quasi experimental methods such as difference-in-difference (DiD), regression discontinuity design (RDD), and propensity score matching (PSM) are examples of how variables based on geospatial data can be leveraged in a GIE.

We will proceed with using the data prepared earlier assuming that we require data at our unit of observation (districts, Ghana ADM2) that contains a record of whether the unit was treated, the change in the percentage of the unit not including water that was cropland (outcome metric), as well as urban (covariate) between 2015 and 2020, the area of the unit, and the region (Ghana ADM1) it is within (fixed effect variable).

First we will import the Python packages needed, and load in the configuration file as done in the previous sections.

```
from pathlib import Path
from functools import reduce

import pandas as pd
import geopandas as gpd
```

```
import rasterstats as rs

from config import get_config

config = get_config()
```

Then we define the paths we will use to load our data, and paths to export the results. Note the usage of the config file to define paths. By using a shared configuration file, the script will automatically generate the same paths used to download the datasets in the previous section. Automating the path generation avoids having to manually enter the paths in multiple locations and some of the potential for human errors. If you were to modify the dataset years when you run the download and processing code, the integration code would automatically adjust.

```
base_path = Path(config["base_path"])

treatment_path = base_path / config["treatment_path"]

adm2_path = base_path / "geoBoundaries" /
f'{config["boundary"]["version"]}_{config["boundary"]["gb_data_hash"]}_{config["bounda
ry"]["gb_web_hash"]}' / "geoBoundaries-GHA-ADM2" / "geoBoundaries-GHA-ADM2.geojson"

adm1_path = base_path / "geoBoundaries" /
f'{config["boundary"]["version"]}_{config["boundary"]["gb_data_hash"]}_{config["bounda
ry"]["gb_web_hash"]}' /"geoBoundaries-GHA-ADM1" / "geoBoundaries-GHA-ADM1.geojson"


raster_items = {f"esa_lc_{year}": base_path / "esa_landcover" / f"esa_lc_{year}.tif"
for year in config["landcover"]["years"]}


output_csv_path = base_path / "output" / "ghana_adm2_data.csv"
output_geojson_path = base_path / "output" / "ghana_adm2_data.geojson"
```


Next we will load both the ADM2 and ADM1 data for Ghana, and perform a spatial join of the ADM1 to the ADM2 based on their intersections. The goal is to determine which ADM1 unit each ADM2 is associated with.

```
# load adm2 and adm1 data
adm2_gdf = gpd.GeoDataFrame.from_file(adm2_path)
adm1_gdf = gpd.GeoDataFrame.from_file(adm1_path)

# drop unnecessary columns
adm2_gdf = adm2_gdf.drop(columns=["shapeISO", "shapeGroup", "shapeType"])
adm1_gdf = adm1_gdf.drop(columns=["shapeISO", "shapeGroup", "shapeType"])

# spatially join the adm1 to the adm2 data
# this will produce rows for every intersection between an adm2 and adm1 polygon
# meaning there will be multiple rows for each adm2 polygon
```

```
adm2_gdf = adm2_gdf.sjoin(adm1_gdf, how="inner", predicate="intersects",
lsuffix="adm2", rsuffix="adm1")
```

After the spatial join, there are many more rows in the new GeoDataFrame than the original ADM2 GeoDataFrame. The additional features are caused by selecting the “inner” join method, which produces a row for every intersection of an ADM2 unit with an ADM1 unit. Since these boundaries are imperfect and even small variations along borders will result in an intersection, there are many intersections resulting for each ADM2 unit. To resolve this problem, we can calculate the actual intersection area for each pair detected and determine which is the most reasonable ADM1 unit to be associated with each ADM2 unit.

First, we need to merge the ADM1 geometry associated with each intersection (row) in order to iterate over the data and calculate the area of the intersection relative to the area of the ADM2 unit. Then we can group the data by ADM2 unit ID, and keep only the row with the larger intersection area.

As a last sanity check, we can make sure that every ADM2 and ADM1 pair selected have an intersection of at least 50% the area of the ADM2 unit. While this is a somewhat arbitrary threshold, it would highlight any major anomalies from the typical relationship across administrative levels.


```
# merge the adm1 geometry into the adm2 data that now has the associated adm1 id
adm2_gdf = adm2_gdf.merge(adm1_gdf[["shapeID", "geometry"]], how="left",
left_on="shapeID_adm1", right_on="shapeID", suffixes=("", "_adm1"))

# drop unnecessary columns from the joins
adm2_gdf = adm2_gdf.drop(columns=["index_adm1", "shapeID"])

# calculate the overlap between all adm2 and adm1 intersections found by the spatial
join
adm2_gdf["overlap_adm1"] = adm2_gdf.apply(lambda x:
x.geometry.intersection(x.geometry_adm1).area / x.geometry.area, axis=1)

# keep only instance of shapeID_adm2 with the largest overlap_adm1
# this should give us the most accurate adm1 polygon for each adm2 polygon since the
boundaries are not perfect
adm2_gdf = adm2_gdf.loc[adm2_gdf.groupby("shapeID_adm2")["overlap_adm1"].idxmax()]

# confirm that we do not have any overlaps that are very small
assert adm2_gdf.overlap_adm1.min() > 0.5, "Overlap is less than 50% for some polygons"
```

For the next step, we need to add the treatment indicator data to the resulting ADM2 GeoDataFrame. This can be accomplished by loading a standard dataframe from the CSV containing the treatment data for each ADM2 unit, and using pandas to merge it with the GeoDataFrame based on the ADM2 shapeID field.

```
# load treatment data
treatment_df = pd.read_csv(treatment_path)

# merge the treatment data into the adm2 data and drop the duplicate shapeID column
adm2_gdf = adm2_gdf.merge(treatment_df[["shapeID", "treatment"]], how="left",
left_on="shapeID_adm2", right_on="shapeID")
adm2_gdf = adm2_gdf.drop(columns=["shapeID"])
```

Next we will aggregate the pixel level raster data to the boundaries of the ADM2 units. As you recall from the previous data processing the land cover rasters are 350 meter resolution, meaning that within each ADM2 there may be thousands of pixels. The values in the land cover rasters are discrete and reflect distinct land cover classes such as urban, cropland, or water. The desired result is the count of pixels within each unit of each land cover class for both the initial year (2015) and the final year (2020) of the evaluation.

```
# set the id field that we will use to merge the results of the zonal stats
id_field = "shapeID_adm2"

# load the category map from the config that is needed for the categorical zonal stats
# the format the category map is saved in in the config is inverted from the format
needed for the zonal stats
# so we need to invert it here
category_map = config["landcover"]["category_map"]
category_map = {v: k for k, v in category_map.items()}
```


Note that the category map we will use for the zonal statistics with the land cover data is different from the mapping used during the processing of the land cover data. The previous mapping was intended to aggregate different pixel values into more generalized categories (e.g., deciduous forest and coniferous forest into just forest). The category mapping is used by the zonal statistics implementation to translate pixel values of categories into human readable fields (e.g., 10 becomes rainfed cropland).

```
category_map.no_data = 0
category_map.rainfed_cropland = 10
category_map.irrigated_cropland = 20
category_map.mosaic_cropland = 30
category_map.forest = 50
category_map.grassland = 110
category_map.shrubland = 120
category_map.sparse_vegetation = 140
category_map.wetland = 180
category_map.urban = 190
category_map.bare_areas = 200
category_map.water_bodies = 210
category_map.snow_ice = 220
```

We will iterate over each raster (land cover in 2015 and 2020) and run zonal stats to extract the data. The results from the zonal stats are turned into a GeoDataFrame and null values occurring due to a particular class not being present in an ADM2 unit are set to zero.

```
# init results list with the existing adm2 gdf
# this will be used to merge the results of the zonal stats
# the zonal stats gdfs will drop the base columns so that they are not duplicated
results = [adm2_gdf]

# loop through the rasters and calculate zonal stats for each
# fill na values with 0
# append each resulting gdf to the results list
for raster_id, raster_path in raster_items.items():
    tmp_stats = rs.zonal_stats(
        adm2_gdf,
        raster_path,
        geojson_out=True,
        prefix=f"{raster_id}_",
        categorical=True,
        category_map=category_map,
        all_touched=True)
    tmp_gdf = gpd.GeoDataFrame.from_features(tmp_stats)
    tmp_gdf = tmp_gdf.fillna(0)
    tmp_cols = tmp_gdf.columns
    # drop columns except the id field and the raster stats to avoid duplicates
    tmp_gdf = tmp_gdf[[i for i in tmp_gdf.columns if i not in results[0].columns or i
== id_field]]
    results.append(tmp_gdf)
```


> Note: For loops are an excellent indicator of code that may be suitable for parallelization or other approaches to optimize and reduce computation time. Later in the chapter we will explore how to parallelize serial code in a for loop using built-in Python functions.

The zonal stats GeoDataFrames contain only the ADM2 ID and the resulting stats. We therefore merge the original ADM2 GeoDataFrame with each new zonal stats GeoDataFrame to produce our main zonal stats output. The result is a set of columns for both 2015 and 2020 where each column represents the count for a single land cover class.

```
# merge results into single gdf
stats_gdf = reduce(lambda left,right: pd.merge(left, right, on=id_field, how='inner'),
results)

# drop the adm1 geometry column
stats_gdf = stats_gdf.drop(columns=["geometry_adm1"])
```

```

| | A | B | C | F | G | R |
|---|---|---|---|---|---|---|
| 1 | shapeName_adm2 | shapeID_adm2 | shapeName_adm1 | treatment | esa_lc_2015_rainfed_cropland | esa_lc_2020_rainfed_cropland |
| 2 | Wassa Amenfi Central | 2480657B10255028610816 | Western Region | 0 | 13367 | 13239 |
| 3 | Tamale Metropolitan | 2480657B1027184338147 | Northern Region | 0 | 3909 | 3811 |
| 4 | Old Tafo Municipal | 2480657B10806173713075 | Ashanti Region | 1 | 12 | 2 |
| 5 | Ayensuano | 2480657B11085167902059 | Eastern Region | 1 | 5389 | 5389 |
| 6 | Dormaa West | 2480657B11503231198633 | Bono Region | 0 | 4002 | 4002 |
| 7 | Agotime Ziope | 2480657B11620193732867 | Volta Region | 0 | 2215 | 2084 |
| 8 | Accra Metropolis | 2480657B11720749290533 | Greater Accra Region | 1 | 1 | 0 |
| 9 | Ablekuma West Municipal | 2480657B12570528382412 | Greater Accra Region | 1 | 1 | 0 |
| 10 | Wassa Amenfi West | 2480657B13145859726554 | Western Region | 0 | 12168 | 12143 |
| 11 | Nkwanta South Municipal | 2480657B13680530894334 | Oti Region | 0 | 3930 | 3627 |
| 12 | Tain | 2480657B14065489870616 | Bono Region | 1 | 4945 | 4879 |
| 13 | Atwima Nwabiagya North | 2480657B14484334363928 | Ashanti Region | 0 | 2233 | 2161 |
| 14 | Atwima Kwanwoma | 2480657B14749544754301 | Ashanti Region | 0 | 2463 | 2238 |

```

Figure X: Output of data integration steps.

One of the last pieces is to utilize all the data that has now been integrated to produce the metrics needed for the GIE. In this example, the outcome and covariate variables needed are the change in cropland, forest, and urban areas. To calculate these, we will first make a copy of the current GeoDataFrame to avoid making any changes (or mistakes) that would require us to rerun the previous steps. While in this example the computation required was relatively light, for GIEs with potentially hundreds of thousands of units of analysis and many datasets being integrated the processing could take significantly longer.

Then we will calculate the total number of pixels across all land cover classes for both years, as well as the total when excluding water bodies. Then we will sum the various cropland categories, before calculating the percentage of each unit represented by cropland, forest, and urban areas. With the percentages for both 2015 and 2020, we can then calculate the change between the two years.


```
```
calc_gdf = stats_gdf.copy()

calc_gdf["lc_2015_count"] = calc_gdf.apply(lambda x: sum([x[i] for i in x.keys() if
"esa_lc_2015_" in i]), axis=1)
calc_gdf["lc_2020_count"] = calc_gdf.apply(lambda x: sum([x[i] for i in x.keys() if
"esa_lc_2020_" in i]), axis=1)

calc_gdf["lc_2015_count_land"] = calc_gdf.apply(lambda x: sum([x[i] for i in x.keys()
if "esa_lc_2015_" in i and "water" not in i]), axis=1)
calc_gdf["lc_2020_count_land"] = calc_gdf.apply(lambda x: sum([x[i] for i in x.keys()
if "esa_lc_2020_" in i and "water" not in i]), axis=1)

calc_gdf["lc_2015_count_cropland"] = calc_gdf.apply(lambda x: sum([x[i] for i in
x.keys() if "esa_lc_2015_" in i and "cropland" in i]), axis=1)
calc_gdf["lc_2020_count_cropland"] = calc_gdf.apply(lambda x: sum([x[i] for i in
x.keys() if "esa_lc_2020_" in i and "cropland" in i]), axis=1)

calc_gdf["lc_2015_percent_forest"] = calc_gdf["esa_lc_2015_forest"] /
calc_gdf["lc_2015_count_land"] * 100
calc_gdf["lc_2015_percent_urban"] = calc_gdf["esa_lc_2015_urban"] /
calc_gdf["lc_2015_count_land"] * 100
```
```

```
calc_gdf["lc_2015_percent_cropland"] = calc_gdf["lc_2015_count_cropland"] /
calc_gdf["lc_2015_count_land"] * 100

calc_gdf["lc_2020_percent_forest"] = calc_gdf["esa_lc_2020_forest"] /
calc_gdf["lc_2020_count_land"] * 100
calc_gdf["lc_2020_percent_urban"] = calc_gdf["esa_lc_2020_urban"] /
calc_gdf["lc_2020_count_land"] * 100
calc_gdf["lc_2020_percent_cropland"] = calc_gdf["lc_2020_count_cropland"] /
calc_gdf["lc_2020_count_land"] * 100

calc_gdf["forest_change"] = calc_gdf["lc_2020_percent_forest"] -
calc_gdf["lc_2015_percent_forest"]
calc_gdf["urban_change"] = calc_gdf["lc_2020_percent_urban"] -
calc_gdf["lc_2015_percent_urban"]
calc_gdf["cropland_change"] = calc_gdf["lc_2020_percent_cropland"] -
calc_gdf["lc_2015_percent_cropland"]
```

Finally, we can export the fully integrated dataset as both tabular data (CSV) without the geospatial features, and as a GeoJSON. The CSV is read for use in subsequent steps of the GIE to run regression analysis or other tasks, while the GeoJSON can be used to visualize the data at the unit of the analysis.

```
# make sure the output directory exists
output_csv_path.parent.mkdir(parents=True, exist_ok=True)

# write output to csv
calc_gdf[[i for i in calc_gdf.columns if i != "geometry"]].to_csv(output_csv_path,
index=False, encoding='utf-8')

# write output to geojson
calc_gdf.to_file(output_geojson_path, driver="GeoJSON", encoding='utf-8')
```


In the next part of the chapter, we will take time to reflect on the various decisions made in the workflow that we just implemented. We will explore what could have gone wrong as well as potential solutions and general opportunities for improvement.

Part 3: Reflection

In the final part of this chapter, we will consider the concepts, tools, and data utilized in the previous application example, and explore potential barriers you may encounter and solutions that may help you to overcome them. Barriers may include computational resource limitations, data storage I/O (i.e., read and write) speed, network bandwidth, processing time, and dealing with the realities of imperfect and changing data providers and datasets. The solutions we present are not universal, but will hopefully provide you with ideas that you can build upon and extend as needed. These include ways to reduce computational resource needs, parallelize processing to speed up workflows, optimize code, and prepare your codebase to be adaptable for future demands. Finally, we will provide you with a number of additional resources to help you progress on your data engineering journey and implement efficient GIEs.

★ Barriers and Solutions

Memory overflows

One of the most common problems we encounter when running data processing pipelines - particularly with geospatial data, and especially on personal computers - is running out of RAM (memory). When a variable is defined in Python, memory is allocated to store that variable for as long as it's needed. This can become a problem when working with large datasets.

In most cases, memory can be conserved by strategically allocating chunks of data to memory at a time. While it can take some extra time to implement, this practice can enable you to run a processing pipeline on a computer that otherwise couldn't. Even if you have the computational resources to load all your data in memory, memory efficient approaches can help collaborators run your code on their computers and facilitate broader reproducibility and replication of your work.

Example: Reading data in chunks

Let's consider the following basic example of opening a GeoJSON file with GeoPandas.

```
```
import geopandas as gpd

# this allocates a lot of memory to fit the big dataset
first_gdf = gpd.read_file("big_dataset.geojson")
```
```

After the `gpd.read_file()` command executes, `first_gdf` is a variable pointing to a large dataset stored in memory. If that dataset is too large to fit in your computer's RAM, or we add additional commands to load other datasets into memory, your operating system will terminate

your program before it finishes running. Sometimes, the fix for this is to simply reorder commands to handle one dataset at a time. Python is a garbage-collected language, which means that it automatically deletes data from memory once it is no longer needed in the program. However, it is possible to explicitly delete a variable using the `del()` command.

When a single dataset is too large to fit into memory, most tools for reading files can be configured to only read chunks of data at a time. Here is how that is accomplished using `gpd.read_file()`:

```
```
import time
import geopandas as gpd
from memory_profiler import profile

download from
https://github.com/wmgeolab/geoBoundaries/raw/main/releaseData/CGAZ/geoBoundariesCGAZ_
ADM1.gpkg
input_path = "/home/userx/Desktop/geoBoundariesCGAZ_ADM1.gpkg"

@profile
def calc_area_full(input_path):
    """
    Read a file and return the geodataframe
    """
    gdf = gpd.read_file(input_path)
    area_full = gdf['geometry'].area.sum()

    return area_full

full_start_time = time.time()
area_full = calc_area_full(input_path)
full_end_time = time.time()

print(f"Total area (without chunks): {area_full}")
print(f"Time taken to read without chunks: {full_end_time - full_start_time:.2f}
seconds")
```
```


Reading the full dataset takes about 0.35 seconds and uses over 350MB. In contrast, using a chunked read with a chunk size of 100 takes approximately twice as long, but uses less than 80MB. Adjusting the chunk size to your application can help optimize the tradeoff between time and memory. For example, if we just a chunk size of 1000 it takes 0.38 seconds and uses about 200MB.

```
```
import time
import geopandas as gpd
from memory_profiler import profile

@profile
def read_file_in_chunks(input_path, chunk_size):
```
```

```
    """
    Generator function to read a file in chunks
    """
    # track chunk index between loop iterations
    i = 0
    while True:
        # print(f"Reading chunk {i}...")

        # calculate slice beginning and end for this chunk
        chunk = slice(i, i + chunk_size)

        # read chunk from input file
        gdf_iter = gpd.read_file(input_path, rows=chunk)

        if len(gdf_iter) == 0:
            # print(f"\tFinished reading all chunks.")
            break

        yield gdf_iter

        # update chunk index
        i += chunk_size


@profile
def calc_area_chunks(input_path):
    # set size of each chunk (as large as possible)
    chunk_size = 100

    area_list = []
    for gdf_iter in read_file_in_chunks(input_path, chunk_size):
        # run processing on chunk gdf
        area_list.append(gdf_iter['geometry'].area.sum())


    area_chunked = sum(area_list)
    return area_chunked


chunk_start_time = time.time()
area_chunked = calc_area_chunks(input_path)
chunk_end_time = time.time()

print(f"Total area (with chunks): {area_chunked}")
print(f"Time taken to read with chunks: {chunk_end_time - chunk_start_time:.2f}
seconds")
```


Example: Raster windowed read/write

An approach we leveraged in our earlier application is windowed reading and writing of raster data. Reading all the data from a fine resolution raster with global coverage can quickly exceed the memory limitations of many computers. To avoid this issue, we can read only specific

subsets of the raster at once. This is useful if you only need a small portion of the raster data for your study area, or if you need to break processing the full raster data into smaller chunks.

Windows can be defined for specific areas, or as in the example below can be automatically generated based on the underlying tiling scheme of the raster data. By using the inherent “block_windows” to iterate over the raster, you can process manageable chunks of data and write them to a new file iteratively until the entire dataset has been completed. Tiling schemes can vary between rasters, so it is always worth checking the tiling format and the shape of each subset returned by the “block_windows” to verify that it will be manageable with the memory available on your system.

```
import rasterio as rio

input_path = "/path/to/input_raster.tif"
output_path = "/path/to/output_raster.tif"

# reading and writing the data in windowed chunks
with rasterio.open(input_path) as src:
    with rasterio.open(output_path, "w", **src.meta) as dst:
        for ji, window in src.block_windows(1):
            in_data = src.read(window=window)
            out_data = func(in_data)
            dst.write(out_data, window=window)


# compared to reading and writing all the data at once:
with rasterio.open(input_path) as src:
    with rasterio.open(output_path, "w", **src.meta) as dst:
        in_data = src.read()
        out_data = func(in_data)
        dst.write(out_data)
```


Example: Generators

Generators are a similar approach to reading files in chunks or using the windowed reading and writing of rasters. Generators are a native Python feature that can help avoid reducing memory usage associated with performing expensive operations like reading a large file into memory. Instead, the file can be opened and read one row or feature at a time, with the reader function “yielding” each relevant piece of the file to its caller.

Generators are a useful tool with a wide range of potential uses beyond reading data, but we will demonstrate their utility when reading geospatial vector data. The below example is particularly useful when dealing with very large vector files, potentially containing millions of features that are too large to store in memory.

```
import fiona
```

```
input_path = "/path/to/input.gpkg

# reading individual features one at a time
def gen_features(input_path):
    with fiona.open(input_path) as src:
        for feature in src:
            yield feature

for feat in gen_features:
       tmp_gdf = gpd.GeoDataFrame.from_features([feat])
       # perform individual processing here


# compared to reading all the features at once
gdf = gpd.read_file(input_path)
# perform bulk processing here
```

Data Download and Read/Write Speeds

Reading and writing data - whether it is when downloading from a remote server, processing data on a cluster, or using outputs for analysis - is one of the most common bottlenecks in data workflows. In short, moving large amounts of data around can be an expensive process (both in terms of the time it takes and potentially financially when using cloud resources) so it is worth extra consideration to make sure the approach you utilize is as efficient as possible.

In Part Two, we explore one use case focused on leveraging localized storage for processing intensive portions of the workflow. The goal was to avoid reading and writing extensively to storage that may be located elsewhere on your network and/or have slower access drives designed more for longer term storage. The location of your processing and the location of the data being used in the processing are important considerations. Whether it is worth copying data to a local drive or accessing directly from a remote location will depend on the specifics of your application. When moving data to cloud storage, always be aware of the transaction and storage costs you may incur and factor those into your broader planning decisions.

If you are downloading large amounts of data, or downloading datasets on a regular basis, it's worth considering how to minimize the time each download takes. There are many potential bottlenecks when downloading data from a provider, including their server's capacity, the ISPs (internet service providers - who you get internet access from) that connect you to each other, and the local network connection from your router to your computer. Use an internet speed test tool (e.g. fast.com) to measure your computer's download speed, and compare that to the speed of your file download. If the file download speed is similar to your overall download speed, you may be able to speed up your connection locally by plugging your computer into

your router directly using an ethernet cable, negotiating with your ISP for a faster download speed, or upgrading your network equipment to take advantage of faster speeds available.

> Note: Download speeds are typically measured in Megabits (Mb) or Gigabits (Gb), which are distinct from Megabytes (MB) and Gigabytes (GB). Be careful to understand the relevant units when comparing transfer speeds.

In many cases, a user's download speeds are much faster than a given data provider's upload speed, in which case there is no way to make the connection faster. In some cases it is possible to open multiple connections to download multiple files simultaneously leveraging concurrency, which can be faster depending on a number of factors. Check your data provider's terms of use before attempting this.

Concurrency is a useful concept that can help speed up tasks, particularly in situations where you have to wait a while between tasks while your processing capacity is at least partially unused. For example, when you make a network request for a resource that a server may take a relatively long time to respond to. By using an asynchronous framework, such as Python's built-in asyncio package, you can continue to advance tasks using the available resources while waiting for other tasks to progress and require resources again.

It is important to note that while concurrency effectively allows you to progress multiple tasks at the same time, it is a distinct concept from actually performing multiple streams of computation at the same time. Concurrency can be implemented with a single process, while multiple processes are needed to truly parallelize your workflow. Async and concurrency can be useful, but can also be complex to implement and manage. While the potential for complexity in parallelization exists, there are very basic approaches that can be extremely powerful when it comes to reducing the overall time to run your workflows. We will explore parallelization in more detail shortly.

There are a few other ways in which downloads might fail, which are worth planning for when designing your data pipeline. Data providers might *rate limit* your downloads, which generally involves limiting the number of files you may download over a given period of time. This can cause your download script to suddenly fail when the limit has been reached. Providers will usually notify you of this limit ahead of time so you can plan accordingly

Downloading files over the internet can sometimes fail in unexpected ways. Your connection might drop, due to a networking glitch or ISP outage. The data provider's servers might go down, or your program could crash unexpectedly. It's a good idea to assume this will happen from time to time, and design your pipeline to handle these issues when they occur.

For these reasons, we recommend your program follow the following steps:

1. Determine which files are needed

2. Check which have already been downloaded
3. Verify that any downloaded files are valid / complete, if possible
4. Download all files that either don't exist yet or did not pass validation

This flow ensures that no matter if you've run your program before or not, the necessary files will get downloaded and verified. It's helpful not to re-download everything each time you run your program, and to check as well as you can whether your files are correct and ready for processing. We leveraged an implementation of this flow in both of the dataset download examples provided earlier, with the exception of validating the status of the downloaded files.

File validation works best when the data provider also includes a file hash (e.g., a ".md5" file) that can be used to confirm the file you have downloaded is identical to the file on their server. Some tools for downloading files also implement their own version of checkpointing, or if supported by the download server, you could implement your own using Range headers within your download requests. A potentially simpler alternative is to record the status of each download in a CSV or other persistent format as it progresses, allowing you to distinguish between files which exist but are incomplete and files which have successfully downloaded.

Parallelization

We are accustomed to our computers doing many things at once: having multiple browser tabs open, as well as a file manager or perhaps a word processor. When you get home and open your laptop, a background process detects your Wi-Fi network and connects automatically. Modern computer hardware is designed with multitasking in mind, allowing for programs to utilize multiple physical compute resources simultaneously.

In the context of a data processing pipeline, it can be highly desirable to accomplish tasks in parallel. Datasets are often broken into thousands of chunks, which can be downloaded or transformed independently. When this type of "embarrassingly parallel" workload becomes too large to accomplish sequentially, a little bit of engineering can quickly pay off in terms of computation time. Python provides many different tools for this class of problem, a few of which we'll touch on here.

The most common computational bottleneck we experience when processing datasets comes down to the CPU. We test our scripts on a subset of the dataset first, but once we scale up the processing times increase significantly. If each chunk takes a few minutes to process, thousands of chunks could take days to complete.

One of the most straightforward methods to handle this problem is to run multiple threads or processes at once. Modern CPUs often have eight or more cores, on which multiple scheduled threads may run at once. By default a Python script is single-threaded, which means that it runs

on one core. However, Python makes it possible to create a pool of workers, each returning their result to the main process when complete.

> Note: We are brushing over some of the finer points of threaded vs. multi-process parallelization here, for the sake of this introduction. At the time of writing multi-threaded workflows are not well-supported by Python, which is why we have chosen to focus on multiprocessing.

Below we provide a simple example of processing a list of numbers using both a standard serial approach with a for loop, as well as the equivalent parallel processing approach using Python's built-in concurrent futures package.

```
```
import time
from concurrent.futures import ProcessPoolExecutor

def task(n):
    time.sleep(1)
    return n * 2

# serial processing using a for loop
serial_start_time = time.time()
serial_results = []

for i in range(5):
    serial_results.append(task(i))

serial_end_time = time.time()
print(f"Serial execution results: {serial_results}")
print(f"Serial execution time: {serial_end_time - serial_start_time:.2f} seconds")

parallel processing using concurrent.futures ProcessPoolExecutor

parallel_start_time = time.time()

with ProcessPoolExecutor(max_workers=5) as executor:
    futures = [executor.submit(task, i) for i in range(5)]
    parallel_results = [future.result() for future in futures]

parallel_end_time = time.time()
print(f"Parallel execution results: {parallel_results}")
print(f"Parallel execution time: {parallel_end_time - parallel_start_time:.2f}
seconds")
```
```

The simple processing function is applied to the list of numbers from zero to four, and sleeps for one second before multiplying each number by two. In serial using the for loop, this takes the expected five seconds to run. However, when running each of the five tasks in parallel the entire list can be processed in one second.

Parallel processing can be an extremely effective tool for reducing processing time, but does have additional considerations. For example, if individual tasks require a large amount of memory, running multiple tasks simultaneously can result in exceeding the available memory on your system. For this reason it is often best practice to test a small subset of your processing prior to parallelization to understand the resource requirements needed.

Another consideration is the overhead associated with creating parallel processes, and is particularly relevant in more broadly distributed systems. If each process must be sent, or read a large amount of data, the cost of doing so may outweigh the benefits of simultaneous processing. In our above example, imagine if it took 3 seconds to send/read the data to be processed, while the processing function itself takes 1 second. In the serial example, with the tasks already loaded and available the entire processing would still take 5 seconds. However, in the parallel processing example it would instead take 4 seconds (3 to load and 1 to run) for each task. Parallel processing would still be faster, but only slightly (5 seconds for serial vs 4 seconds for parallel), and may not justify the added computational complexity.

Code optimization

Code optimization is often one of the most improperly utilized approaches for improving programmatic workflows. Typically, this derives from programmers focusing on arbitrary elements of a code base that they *think* are slow - often at the expense of readability and maintainability - rather than first testing and analyzing (also known as profiling) their code to see where bottlenecks exist. In many cases, bottlenecks can be addressed through approaches such as those we have discussed above rather than optimizing specific segments of code.

Various tools such as Python's "cProfile" and "memory_profiler" can facilitate profiling your code and identifying specific functions or lines responsible for taking a long time to run, being called frequently, or consuming large amounts of memory. Once you have profiled your code and identified a bottleneck you can then explore approaches for addressing or optimizing as needed. Always consider the broader context when evaluating profiles and potential bottlenecks. If your code is run once a month and takes thirty minutes to run, that may seem long while you are sitting there waiting for it to finish, but in reality that is likely to be an inconsequential amount of time and does not warrant optimization. Similarly, if your code takes ten minutes to run and the slowest segment of code takes one minute, consider whether reducing that even by 90% would really improve your application.

The most significant uses of code optimization typically occur when short, specific functions (e.g., a single line of code performing a mathematical operation over an array) have an exaggerated impact on your application. One example of this dealing with geospatial data is performing statistical functions over arrays representing raster data. Array operations written in pure Python have the potential to be inefficient, and raster calculations in an otherwise simple workflow can take up a significant portion of the processing time (and memory). Fortunately, packages such as numpy implement wrappers for array functions written in far more efficient and quicker programming languages. Often, finding optimization in these cases is simply a matter of finding the right existing implementation of the functionality you need. And more often than not, someone has solved a similar problem to you in the past and searching online through sites such as StackOverflow and GitHub can reveal a wealth of resources.

Computational limits

Even after leveraging all of the above approaches to make your workflows more efficient, you may encounter scenarios where your local computer does not have the resources to process particular datasets in the way your GIE requires, or doing so would take an unreasonably amount of time. In these scenarios you could either upgrade your computer, or leverage other computational platforms that may be available to you. Alternative platforms include a university or research center high performance computing (HPC) cluster, various cloud infrastructure providers (e.g., AWS, Azure, Google), or application-specific platforms such as for processing geospatial data (e.g., Google Earth Engine).

While in some instances it may be sufficient to use these alternatives simply as a more powerful computer on which to run your code, much of the true power of these platforms comes from leveraging distributed computing. Whereas basic parallelization involves splitting your task across multiple processes on one machine, distributed computing splits your task across many processes on many machines in order to greatly increase the potential resources available.

Setting up a distributed computing environment can be an extremely complex task requiring substantial expertise, but fortunately infrastructure and tools exist that can abstract much of the underlying technical complexity and allow you to focus on optimizing your application. The specific implementation approaches and how you modify your code will vary from platform to platform but the underlying concepts of distributed computing are essentially the same as parallelization.

If you are working in a traditional HPC environment often used in academic or research institutions, you should familiarize yourself with the job scheduler provided by your system administrator. This might be SLURM, which allows you to submit commands to be run on a cluster as soon as resources become available. These systems often allow you to use MPI, or Message Passing Interface, to efficiently transfer data and otherwise communicate between

processes running on separate computers. The Python package mpi4py allows you to utilize an existing MPI cluster with relative ease.

Kubernetes (often referred to as “K8s”) has emerged as a popular orchestration solution for compute clusters, and is particularly common among cloud providers. It provides a complex yet versatile set of virtual resources that you can use to run complex applications, including parallelized data processing workloads. We love Kubernetes for all it can accomplish, but recommend against using it unless you can justify the development overhead. There is extensive documentation available online if you’d like to learn more.

Another tool of note is Dask, which makes it relatively easy to orchestrate Python workloads across a cluster of computers. You can spin up a Dask scheduler on a primary node, and workers on the rest. The scheduler will automatically distribute tasks to each worker as you submit them. Dask is particularly convenient as you can write Dask-ready code that can easily be adapted to a ephemeral local “cluster” or to a large-scale distributed cluster. A range of packages and tools are available to facilitate integrating Dask with other tools and systems, and to facilitate administering a Dask cluster.

Various cloud services for distributed computing are available from major providers such as Amazon, Google, and Microsoft. Whereas on a traditional “on-premises” cluster such as a university HPC must be managed entirely by you (or your IT team), different types of cloud services are available which abstract certain aspects of the administrative elements. For example, Infrastructure as a Service (IaaS) allows you to essentially use the provider's raw hardware. With IaaS you still need to manage the operating system, databases, dependencies, application code, scaling, and more. Platform as a Service (PaaS) is effectively a ready made environment for deploying your code. PaaS allows you to configure your data and application but handles most of the technical components of making the distribution work. One of the most simplified options is Software as a Service (SaaS) which typically involves a pre-built application designed for a specific task. You load your data and sometimes code, and the SaaS takes care of deployment, scaling, and most other elements behind the scenes. There are various permutations and differing implementations of all of these types of cloud services, but all involve paying for the convenience and availability of cloud hosted resources.

IaaS is most likely to be used in a GIE by an organization with IT and technical capacity, but a lack of dedicated hardware. Examples of IaaS are typically the core platforms of major cloud providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Compute Engine (GCE). With IaaS you could incorporate SLURM, Kubernetes, Dask, or many other options to implement your cloud based cluster.

PaaS are generally geared more towards applications (e.g., a website) than facilitating computations and analysis. While a PaaS is unlikely to be what you use to conduct your GIE, it may be valuable in sharing the data, analysis, or facilitating other engagement with your work.

SaaS is the most accessible and typically aims to facilitate use without any knowledge of distributed computing. You're likely already familiar with SaaS - examples include most email services (e.g., Gmail), content streaming services (e.g., Netflix), and many more consumer services. One of the most relevant SaaS for a GIE is Google Earth Engine (GEE). GEE is a service entirely focused on utilizing satellite imagery and geospatial data. The service is primarily browser based and integrates a coding environment and visualizations. Users can load their own data as well as access a petabyte scale catalog of data. GEE applications are written in Javascript and use a large library of built in functions to leverage Google's distributed computing capabilities. There is also a Python package and API that allows you to use Python code instead of Javascript for GEE.

While GEE can be an excellent option for aspects of GIEs - particularly certain imagery processing - and is free for academic and research use, it does have limitations. Two of the most notable limitations are the framework of geospatial functionality and processing itself, as well as importing and exporting data. You can write custom functions in GEE, but are in many ways restricted to processing modalities that are chosen for you (known as map and reduce). This restriction is a part of what makes the scaling and processing power behind GEE possible, but can be difficult to work with depending on your goal. Additionally, GEE functions best when working with datasets within the existing environment. If you need to import gigabytes of imagery you acquired from another source not connected to GEE, you may need to pay for storage and related fees. Similarly, if you are processing gigabytes of imagery and would like to export large rasters for use outside of GEE, it can be very difficult to do so. If you are considering GEE as part of your GIE workflow, we recommend exploring it in further detail to determine if it will work for your application.

There are many options for distributed computing and cloud services in particular. This section provided a brief introduction to some of the options available, but is by no means exhaustive. If you need greater computational capacity and have grown beyond what you or your organization can host yourself, we recommend a thorough assessment of the various options available before making a decision.

Preparing for the future

Few things are truly future proof, particularly when it comes to data and technology. However, good data engineering and programming practices can go a long way towards ensuring that the time and effort you spend writing code and developing data pipelines for one project will continue to provide value to you (and hopefully others) in the future. Fortunately, the practices we have introduced geared towards making code and workflows reproducible are closely aligned with those that make code extensible and adaptable to future projects. To build on this, we will consider additional practices and expand on existing concepts to help you get the most out of your time spent creating data workflows.

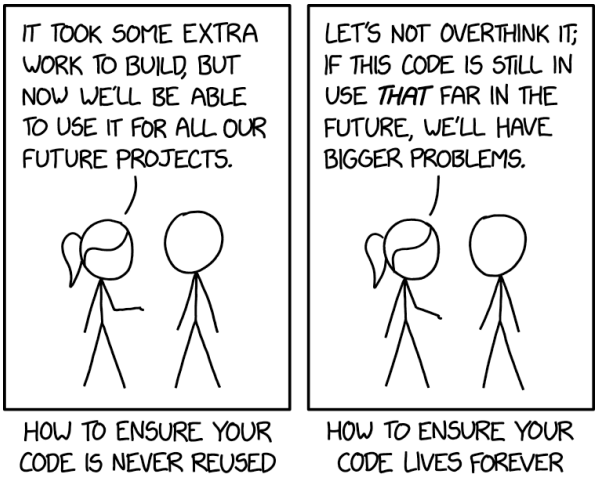

Figure X: A humorous take on the effort to produce reusable code. https://xkcd.com/2730/

One of the best pieces of advice we have for producing reusable code is to create good documentation. In-line comments that explain each function or dense line of code make it easier to revisit a codebase, understand the purpose, and reuse or adapt components for future work. Documentation will help you in the future as much as it will help others, but to facilitate sharing code and documenting your process, we encourage you to share your code publicly leveraging resources such as Git and GitHub - preferably using an open source license to allow others to benefit from your work.

There is often a point when progressing from a rough proof of concept to an operational pipeline, that what started off as a single function or script becomes too long and unwieldy to understand. To avoid this, we recommend breaking different functionality out into modules and functions, each with well-defined roles within a larger system. This makes your codebase more approachable when a new feature is required, or to write tests that verify individual components. There are numerous approaches for organizing files, functions, and other elements of a codebase which we encourage you to experiment with over time. Exploring approaches used by other projects is a great way to find one that works best for you.

One of the more frustrating aspects of data pipelines is when data providers change some aspect of how you access the data. Potential changes include the data format or file structures used to store the data, authentication mechanisms for downloading data, shifting repositories entirely, as well as introducing new caveats or data usage considerations. Unfortunately, it is difficult to rely on even well-established organizations to provide the same data twice, especially after months or years. If you can, we recommend storing a copy of not just your pipeline's output, but the raw download from the source. This helps ensure that you can replicate your own work down the line. In the event that a data provider does change and you need to download new data (e.g., the latest year or a new version with better accuracy) modular and extensible code with good documentation will often allow you to adapt your existing code without significant issues.

Another element of dealing with data provider changes is recognizing there is an issue in the first place. In many cases data pipelines may run on an automated schedule or will not be completely reviewed and retested before being used to download new data. To account for this, your code should aim to “fail fast:” as soon as something is not right, it should throw a clear error and refuse to proceed. It’s a good idea to add redundant checks at the beginning and end of functions to verify that the variables you are passing contain the type of data you expect and certain assumptions about the data are met. Python’s “try-except” blocks, “assert” statements, warnings, and “Exceptions” classes are useful tools for identifying and flagging issues.

Similar to how unit tests must be designed with your specific code/functions in mind, testing data is highly specific to each application. One example to consider when working with raster data is the expected shape of each layer in the dataset. If you are downloading twenty years of rasters representing land cover classification at a consistent extent, resolution, projection, and creation methodology then it is likely that the dimensions of each layer should be consistent. During your download or processing workflow, it is worth checking the dimensions of each layer to verify they align. Preemptively identifying an issue here could help prevent a more difficult to diagnose and manage error in later processing or analytical steps.

Over time you will become more familiar with common sources of error that you can test for in your code, as well as learn how to best mitigate them when they arise. Even the most experienced coders encounter new issues and bugs, but the best coders learn from each new project and use that knowledge to make teach subsequent project a little better.

Alternative programming language

While we’ve focused heavily on Python in this chapter, there are many other languages that may be suitable for your GIEs and data workflows. Python has become particularly popular for data science, automation, and related workflows, but like all languages it has pros and cons. The readability and accessibility of Python is often contrasted against its computational speed. Compiled languages such as C can be orders of magnitude faster than Python, but can take far more time to develop. Writing geospatial analysis logic in compiled languages comes at a cost: lower-level code is often more complex, and requires a more advanced understanding of the data structures and algorithms at play. Ultimately, it is up to you to decide what language is right for your work based on the language(s) you already know (learning a new language is not trivial), what your colleagues use, and how well suited to your task it is.

GIEs do not typically involve workflows that must be run immediately and frequently and would require an extremely fast compiled language. However, specific aspects of your code that involve complex processing may justify leveraging a faster language. C++ is one of the most popular compiled languages, along with other well established languages such as Java or C. Rust and Go and newer languages which offer many improvements and speed. While all of these do generally have a steeper learning curve if you don’t already know them, they can also

be leveraged alongside easier to learn languages such as Python to implement specific portions of code. Many Python packages are actually wrappers for code written in C or similar languages, which allows you to use Python while still getting much of the benefit of a faster language.

Two popular languages among disciplines commonly involved in GIEs (e.g. economists) are Stata and R. Both have established themselves in large part for the availability of packages and libraries which implement statistical models and related functions. Neither of these are particularly fast or suited to developing automated applications, but their frequent use in curriculums and research organizations means they are a common language for many. The ability to share and allow others to understand or contribute to your code is a significant consideration that may justify computational speed or other features.

Most programming languages have reasonable support for geospatial processing and analysis, though Python has one of the most active geospatial communities. However, most of the core geospatial functionality available in Python packages is really derived from geospatial packages originally developed for C++, Java, or other languages. Be sure to explore the geospatial ecosystem for any language before committing to it, and make sure it meets your needs.

In short, if you are already familiar with using a specific language for GIEs or working with geospatial data, go with that! If you are new to programming, Python is a good choice because it has a relatively gentle learning curve, and offers one of the largest and most active ecosystems of packages and community support. If you find yourself reaching the limits of Python and needing more fine-grained control of your computer's resources, the community of high-performance geospatial library maintainers will welcome you with open arms.

★ Resources

The following is a brief list of resources to supplement those provided throughout the chapter. The resources we list here and throughout the chapter are not exhaustive or necessarily reflective of what is "best" for your particular application. The aim of these resources is to provide you with a starting point to begin your own data engineering journey and dive into the broader ecosystem after finishing this chapter. It is up to you to explore these and other resources and gauge their suitability for your particular application. Keep in mind that data and tools, like most things in life, will change over time. We hope this list of resources proves useful for you, but most important is that the underlying concepts and skills introduced in this chapter will apply and serve you regardless of the specific data and tools you may work with in the future.

Data

- NASA / USGS operate a wide range of programs with varying satellites, sensors, missions, and data products. Historically, datasets have often been hosted in various project specific silos, but the majority is now being migrated to the Earth Science Data Systems (ESDS) Program. Existing DAACs (Distributed Active Archive Center) used to distribute NASA's data from various programs include the LAADS DAAC (Level-1 and Atmosphere Archive & Distribution System). The USGS EarthExplorer has been another platform for accessing a range of imagery products.
- The European Space Agency (ESA) also has a range of valuable satellites and products. These can primarily be accessed through their main data catalog, the climate data store (CDS), and the Copernicus Data Space Ecosystem (CDSE).
- Until recently, the Center for Integrated Earth System Information (CIESIN) out of Columbia University operated the NASA Socioeconomic Data and Applications Center (SEDAC). SEDAC has developed or distributed a wide range of socioeconomic datasets. Due to a stop work order from the government, the SEDAC data is currently only available through a spreadsheet. This may change in the future.
- The geoBoundaries administrative boundary dataset used in examples during the chapter is a free and open source collection of global boundaries.
- OpenStreetMap (OSM) and Geofabrik
- GeoQuery -
- Humanitarian Data Exchange (HDX) -
- Harvard Dataverse -

Coding tools, resources, documentation

- AI/LLM coding tools are an increasingly prevalent resource for developing code and applications. The generic ChatGPT can be used for coding, while coding specific tools exist as well. One popular example is GitHub Copilot which can be integrated with VS Code. AI coding tools are a very new and evolving set of resources that should be

carefully considered when incorporating into your development. AI tools can be very helpful when used to supplement coding (e.g., code completion, generating boiler plate code) but also run the risk of causing more work or introducing bugs if relied on to create applications entirely.

- Pytest provides a useful framework for developing unit tests in Python.
- Ruff is one option for a linter in Python. There are other tools for linting and formatting with various differences such as flake8, pylint, and black.
- The Official Python Documentation is an invaluable reference when working with Python code. A key strength of Python is the expansiveness of its built-in library, and how well documented it is.
- Python's PEPs (Performance Enhancement Proposals) are a great resource for diving further into a range of topics at the heart of Python. PEPs range from guiding principles of Pythonic programming, to style guides, technical implementations of features, and future plans.
- QGIS is an excellent free and open source option for desktop GIS software. It can be used to visualize and explore geospatial data, as well as conduct a range of basic analysis with a relatively easy to use desktop interface. The official documentation is invaluable, and there is an extensive community which produces tutorials and answers questions across various platforms such as StackOverflow.
- ArcGIS is a proprietary web based and desktop GIS application that can serve the same functionality as QGIS and potentially more (e.g., web hosting maps and other geospatial data visualizations). Some organizations may already use and pay for ArcGIS. See ArcGIS Pro (desktop software), ArcGIS Online, StoryMaps, and more.
- OSGeo or the Open Source Geospatial Foundation supports a range of open source geospatial projects and related efforts. Projects they support include GDAL, QGIS, PROJ, and PostGIS. They also provide a range of resources, and facilitate events such as the FOSS4G annual conference.

Further reading, advanced topics in data engineering and programming

- *The Missing Semester of Your CS Education* are online course materials published by professors at MIT. It covers many of the fundamentals of developing software and working with data, including git, terminal commands, and code editors.
- Harvard CS50's Introduction to Programming with Python
- University of Helsinki's Python Programming MOOC 2023
- Geocomputation with Python
- Geographic Data Science with Python
- Introduction to Python for Geographic Data Analysis

Conclusion

In this chapter we covered a range of topics and best practices related to effectively implementing data engineering concepts into your GIE data pipelines and workflows, with a particular emphasis on reproduction and replication. As you internalize effective data engineering paradigms and integrate them into your daily work, you will be establishing a foundation that will continue to support you throughout the life of your projects. Incorporating well documented and extensible code that follows programming standards will almost inevitably result in improvements in code quality, less technical debt (the cost of maintaining code in future), and lower development costs for future projects (both for you and others) that can reuse existing code.

Although we introduced a specific set of tools to facilitate the examples in the chapter, the fundamental concepts and approaches are almost entirely transferable to your own choices and preferences. If you are just getting started, the options we presented (VS Code, GitHub, and Python) are flexible and accessible tools that can continue to serve you just as well as a beginner as when you have years of experience. As you grow and create more advanced data engineering workflows you may continue to use this chapter as a reference, and adapt the examples to your needs.

The topics we covered were not exhaustive, but the chapter aimed to provide you with enough knowledge to help you work through the range of unforeseen challenges you may encounter in real work situations on your own. You should be prepared to take these lessons into your next GIE and start to incorporate them. If you are feeling overwhelmed, know that even incorporating one or two elements will make a difference. With each new project and each concept that eventually becomes routine for you, your data workflows will become better and better. Improvement does not happen overnight, but is a gradual process that is well worth the effort.

Data is at the heart of every GIE, and by improving your data engineering skills and practices you can help drive more reliable and efficient evaluations. With increasingly limited budgets and resources for evaluations of development efforts, it is critical to use what resources we do have as effectively as possible. While the unique nature of data for any individual project can present challenges in streamlining workflows, by approaching your work with the mindset of a data engineer, you can make implementing GIEs a more manageable process.

References

1. Goodman, S. et al. (2019) GeoQuery: Integrating HPC systems and public web-based geospatial data tools. Computers & geosciences Volume 122 Volume 122, Pages 103-112 https://doi.org/10.1016/j.cageo.2018.10.009

2. Runfola, D. et al. (2020) geoBoundaries: A global database of political administrative boundaries. PLoS ONE 15(4): e0231866. https://doi.org/10.1371/journal.pone.0231866

3. Munroe, Randall. XKCD. License: https://xkcd.com/license.html