Data Engineering, Data Science, & DevOps [DRAFT]

Stephen Morgan

06/10/2020

Table of Contents

This document attempts to outline key principles, best practices, tools, and strategies to aid in the Data Engineering, Data Science, and DevOps. The purpose and scope of this document is to clearly outline organizational goals for data science in the short, mid, and long-term time frames in order to advance the core data analysis and data science capabilities to support client mission.

The first sections of this document focus on data engineering: data descriptions, data requirements, databases, and data warehousing. These sections are then followed by topics related to data science development strategies and operations, including: standards and best practices, virtual environments, and test driven development. Lastly, this document presents information and strategies on DevOps: CI/CD, infrastructure, microservices, automation, cloud migration, etc.

Many of these sections overlap and are closely tied to one another. While this document attempts to decouple many of these relationships in order produce a manageable road map, it is impossible to completely isolate each section.

There are numerous benefits of having this document:

* Technical reference for staff and client
* Planning and vision road map for management
* Source of content for re-competes
* Requirements for staffing & resources requests

# Data Engineering

## Why it matters: TLDR

Data Engineers are data professionals who prepare the data infrastructure to be utilized by Data Scientists. They are often software engineers, although sometimes they are data scientist who have transitioned into data engineering. Data Engineers are the ones who design, build, integrate data from various resources, and manage big data. They can also write complex queries on top of the data, making sure it is easily accessible, works smoothly, and is optimized for the performance of their company’s data ecosystem. They might also run some ETL (Extract, Transform and Load) on top of datasets and create data warehouses that can be used for reporting or analysis by data scientists.

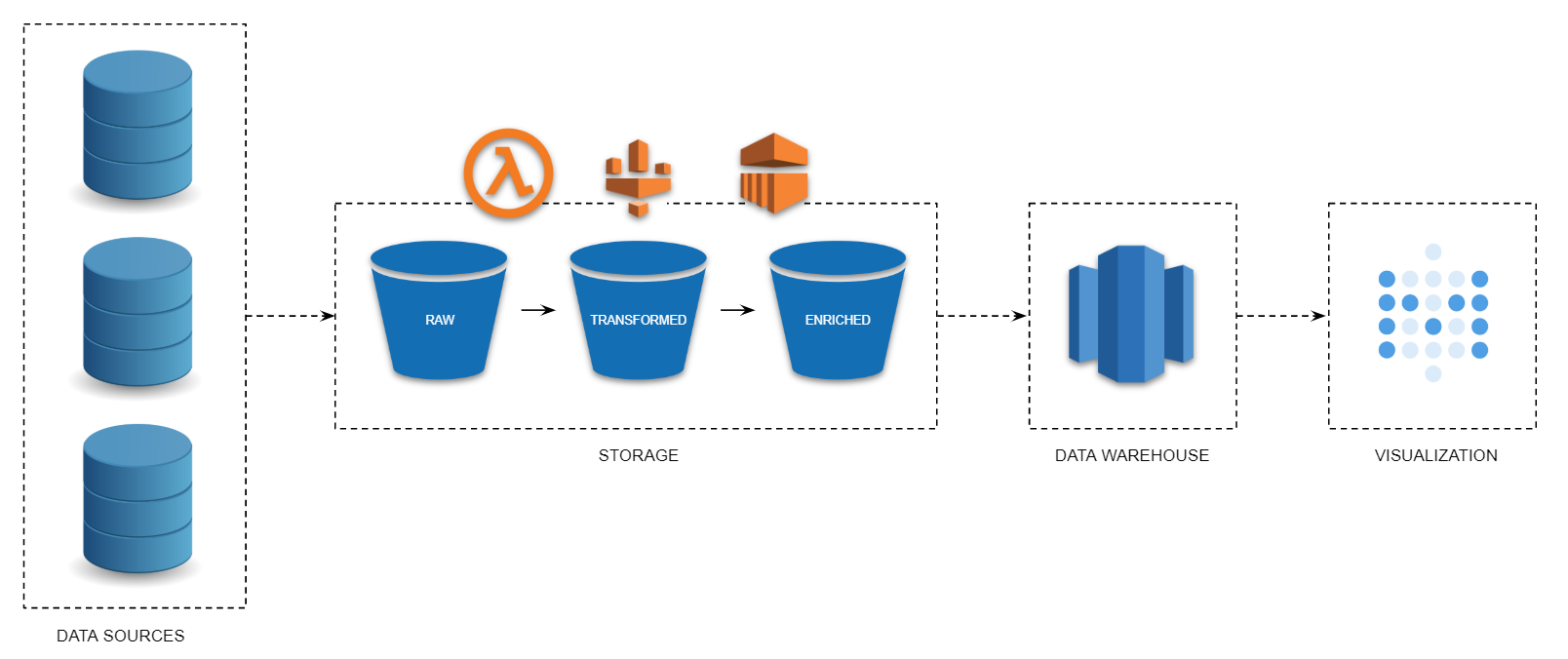
Broadly speaking, data engineers fall into a few categories, depending on the type of company or work they perform:

* Generalist
* Pipeline-centric
* Database-centric

A generalist data engineer typically works on a small team. When a data engineer is the only data-focused person at a company, they usually end up having to do more end-to-end work. For example, a generalist data engineer may have to do everything from ingesting the data to processing it to doing the final analysis. Unless already familiar with machine and deep learning modeling, this requires more data science skill than most data engineers have. However, it also requires less systems architecture knowledge — small teams and companies don’t have a ton of users, so engineering for scale isn’t as important.

Pipeline-centric data engineers tend to be necessary in mid-sized companies that have complex data science needs. A pipeline-centric data engineer will work with teams of data scientists to transform data into a useful format for analysis. This entails in-depth knowledge of distributed systems and computer science.

A database-centric data engineer is focused on setting up and populating analytics databases. This involves some work with pipelines, but more work with tuning databases for fast analysis and creating table schemas. This involves ETL work to get data into warehouses. This type of data engineer is usually found at larger companies with many data analysts that have their data distributed across databases.



Data Warehouse Pipeline, AWS

## Data Description(s)

TODO:

* Expand section
  + Add descriptions of how Team utilizes data
  + Add general information describing the data itself

## Data Requirements

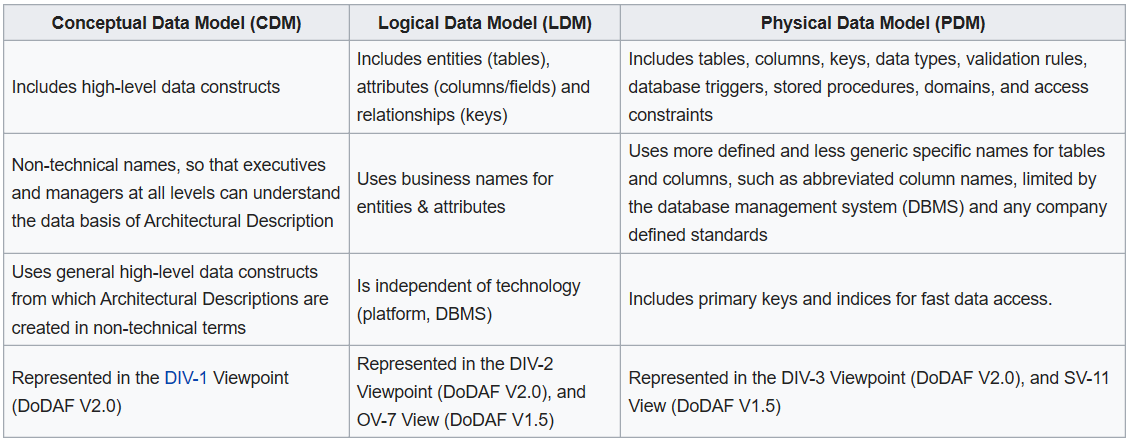
TODO:

* Expand section
* Develop Data Requirements for existing reports, dashboards, and models
* Schedule stakeholder interviews to capture data requirements:
  + What are the current data goals & objectives for the team, for each application?
  + Where does data live now?
  + What is the process for gathering and processing?
  + What applications (reports, models, dashboards) use this data?
  + What is the required data quality and structure?
  + Who has access, what is the process of getting access?
  + How to deconflict overlapping users/access?
  + Future State – Data Lake/Warehouse

Data Requirements, a sub-process of Data Management and typically performed by data quality analyst(s), focuses on the information needs of data-consuming applications and the departments and teams that develop and maintain them, providing a standard set of procedures for identifying, analyzing, and validating data requirements and quality. Data Requirements are intended to help:

* Articulating a clear understanding of data needs of data-driven processes
* Identifying data quality dimensions associated with those data needs
* Assessing quality and suitability of candidate data sources
* Aligning and/or standardizing the exchange of data
* Implementing production procedures which evaluate data conformance and coerce data in the production flow
* Reviewing and identifying improvement opportunities in relation to downstream data need
* Project, contract, or staff transitions
* Designing and developing a common data(base) source & pipeline

Data requirements are often an application-centric document which is used to document the required data (and its structure and quality) for a model, report, or dashboard. This type of data requirements document is beneficial for transitions of team members and recording business objectives. However, an analysis of data requirements can aid in designing and developing appropriate data models, which in turn can be utilized in developing shared databases and pipelines across teams and departments.



Data Models

## Databases

Just because csv files fit in memory, it doesn’t mean that you’re better off than using a SQL database. In fact, you may be handicapping your Machine Learning efforts unnecessarily by ignoring your SQL RDBMS as a data management platform. In most cases, the performance benefits from connecting to data stored in a database are greater than loading data into memory. As a general rule, vectorized operations are going to be more efficient in R and row-based operations are going to be better in SQL.

Beyond just the performance benefits, there are other important reasons to use a database in a data science project. SQL is not a machine learning tool by any stretch, but it’s possibly the most powerful tool available to manage the data you need for your Machine Learning projects. *If you’re moving towards operationalization, then it becomes essential to use one*. R and Python are top class tools for Machine Learning and should be used as such. While these languages come with clever and convenient data manipulation tools, it would be a mistake to think that they can be a replacement for platforms that specialize in data management. Let SQL bring you the data exactly like you need it, and let the Machine Learning tools do their own magic.

Let SQL do the heavy lifting when you need to sort, filter, join, group, compute built-in aggregates using SUM, AVG, STDEV, MIN, MAX, and COUNT. In other words, use SQL to perform common types of Business Intelligence (BI) type analysis. It’s a mistake to try to use R to do things that database engines excel at. Anytime you’re looking towards the past to understand what already happened, you’ll want to compile aggregates, filters, and joins from historical data to form a picture of a state at a period in time. *BI tools have been perfecting this since the early 90’s, so it makes little sense to try to replicate that capability with R now*. The database engine should be seen as a way to offload the more power-hungry and more tedious data operations from R or Python, leaving those tools to apply their statistical modeling strengths. This division of labor makes it easier to specialize your team. It makes more sense to hire experts that fully understand databases to prepare data for the persons in the team who are specialized in machine learning rather than ask for the same people to be good at both things.

Where R and Python shine is in their power to build statistical models of varying complexity which then get used to make predictions about the future. It would be perfectly ludicrous to try to use a SQL engine to create those same models in the same way it makes no sense to use R to create sales reports. There’s really no point in using a Machine Learning tool (aka R or Python) to tell you what the percent difference in sales is between this quarter and last quarter; *let SQL do that*. Use R or Python when you need to perform higher order statistical functions including regressions of all kinds, neural networks, decision trees, clustering, and the thousands of other variations available. Use SQL to retrieve the data just the way you need it. *If you find you’re still doing cleanup work on your data using R after retrieving it from the database, then you’re under-utilizing SQL*. Then use R or Python to build your predictive models. The end result should be faster development, more possible iterations to build your models, and faster response times.

A perfectly valid, and recommended, strategy for analyst is to fine tune the SQL queries to extract data at a manageable size for your PC’s memory either by filtering, aggregating, or sampling, then import into R instance in memory so that you can quickly and iteratively explore and analyze the data with all the statistical horse powers.

#### When to use a database

In short, this is when you need to be using a database instead of flat files:

1. Server-side computing

* If your data is going to be made available to a team of 2 or more people, you’ll want to be sure that your infrastructure can support concurrent access to your data; something relational database engines are especially good at. Scaling from 2 to several thousand users is not an issue. You could put the file on a server to be used by R Shiny or ML Server, but doing makes it nearly impossible to scale beyond few users. As an example, one 30 gigabyte dataset on the shared server will load separately for each user connection. So, if it costs 30 gigabytes of memory for one user, for 10 concurrent users, you would need to find a way to make 300 gigabytes of RAM available somehow.

1. Scalability

* The above example uses a 30 gigabyte file as an example, but there are many cases when data sets are much larger. In financial institutions, research organizations, and telecoms, it’s possible to be working with several terabytes of data at a time. This is easy work for relational database systems, many which are designed to handle petabytes of data if needed. For the purposes of data science, you’ll need the ability to query subsets of data as well as aggregates computed on the server in a way that would be impossible on a server.

1. Clean data source

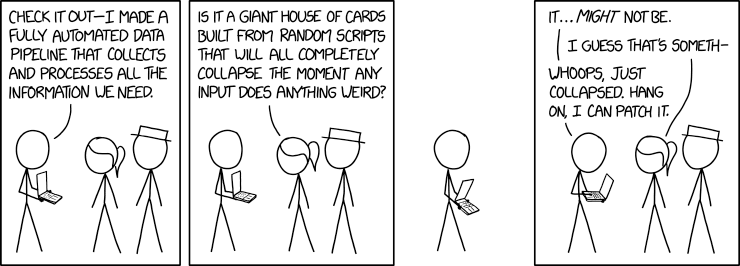
* We commonly need to perform cleanup of data files such as converting strings to dates, removing “None” from numeric columns, and removing a few records that seemed corrupt. We also routinely add columns for analysis or models by performing conversions of numeric values, such as dates or times like “YTD” and “FY”, or averages like “ADP” and "ALOS etc. This may be a time-consuming operation that would be good to perform once and then store the results so that you and other team members can be spared the expense of doing it every time you want to perform your analysis.

1. Security

* Most RDBMS engines come with robust security features that make it possible for data to be read-only for some users, inaccessible to others, and read/writable for super-users. If you’re working with confidential data, you can make use of more advanced security features that block access to portions of information or even obfuscate specific columns that contain things such as NHS or SSN numbers.

See details here: <https://www.r-bloggers.com/r-and-data-when-should-we-use-relational-databases/>

## Data Pipelines



xkcd comic, data pipeline

Sometimes the terms ETL and data pipeline are used interchangeably, but they are not the same process. **ETL** stands for Extract, Transform, and Load. ETL systems **extract data from one system**, transform the data and load the data into a database or data warehouse. ETL pipelines usually run in batches, which typically occur in regular scheduled intervals.

On the other hand, **data pipeline** is a broader term that encompasses ETL as a subset. A data pipeline is a set of actions that **extract data (or directly analytics and visualization) from various sources**. It is an automated process: take these columns from this database, merge them with these columns from this API, subset rows according to a value, substitute NAs with the median and load them in this other database. This is known as a “job”, and pipelines are made of many jobs.

In pipelines the data may or may not be transformed, and it may be processed in real time (or streaming) instead of batches. When the data is streamed, it is processed in a continuous flow which is useful for data that needs constant updating. In addition, the data may not be loaded to a database or data warehouse. It might be loaded to any number of targets, such as an AWS bucket or a data lake, or it might even trigger a webhook on another system to kick off a specific business process. Data pipelines can also be a web service for processing and moving data between different (AWS) compute and storage services, as well as on-premises data sources, at specified intervals.

Generally speaking, data pipelines are for teams that:

* Generate, rely on, or store large amounts or multiple sources of data
* Maintain siloed data sources
* Require real-time or highly sophisticated data analysis
* Store data in the cloud

## Data Lakes (Warehouse)

TBD

# Data Science

## Why it matters: TLDR

TBD

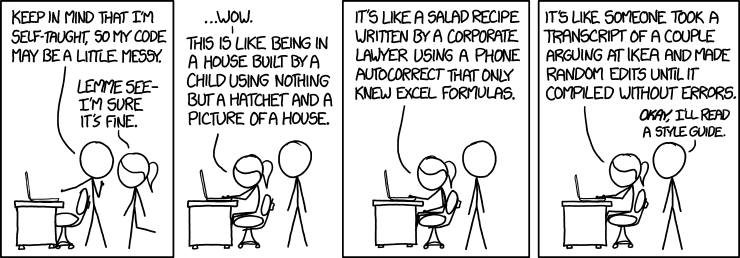
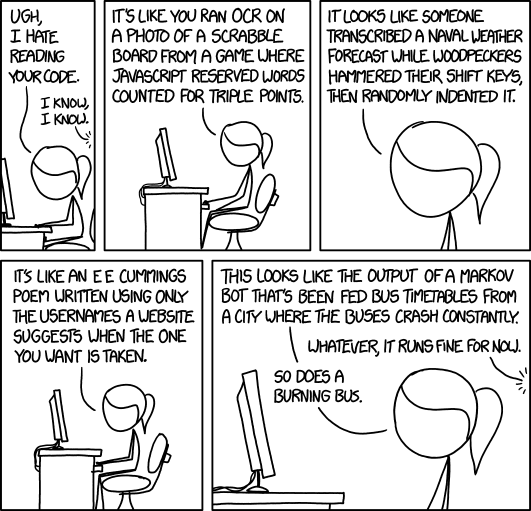
## Standards & Styles

“A foolish consistency is the hobgoblin of little minds” — Ralph Waldo Emerson

### Clean Data

TBD

### Clean Code

All style guides are fundamentally opinionated. Some decisions genuinely do make code easier to use (especially matching indenting to programming structure), but many decisions are arbitrary. The most important thing about a style guide is that it provides consistency, making code easier to write because you need to make fewer decisions. Reducing technical debt along is worth it’s weight in gold, and there’s plenty of [technical debt in ML research](.//assets//pdf//hidden-technical-debt-in-machine-learning-systems.pdf)

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important. — PEP 8

Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides take precedence for that project.

However, know when to be inconsistent – sometimes style guide recommendations just aren’t applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. — PEP 8

There are numerous python packages for linting and styling, most of which adhere to PEP 8 standards. There are fewer R [linters](https://github.com/jimhester/lintr) and [stylers](https://styler.r-lib.org/), but RStudio comes stock with tools to help with basic formatting and styling.

Regardless of which language Data Analysts and Scientists utilized, there are baseline code standards that can be adopted. For instance, standard descriptive information, or script metadata, is typically included at the very beginning of all scripts. This descriptive section is where a line item indicating SENSITIVITY, which would denote if the script uses sensitive information. Following the script metadata, it is common to include all libraries/packages. For less common libraries, it is beneficial to include short descriptions of what the library does or is intended to do.

Following library imports, it would be beneficial to include all input and output pathways so that files and directories can be easily found and edited. Another common standard is to use all upper case when sectioning off code within a script, which makes identifying sections easier. Each of these standards can be easily included in a basic template and implemented into code bases. An example tying the above together is as followed, which utilizes RStudio’s sections functionality:

# META --------------------------------------------------------------------  
  
 #PURPOSE: Why the script exists and the purpose for developing the script  
 #OUTPUT: The expected output from running the script  
 #AUTHOR: Name of person who owns/maintains/created the script  
 #DATE: Date of Creation  
  
 # READY ENVIRONMENT --------------------------------------------------------  
  
 # for science  
 library(caret)  
 # for getting certain packages  
 library(devtools)  
 # for forward-pipping in R  
 library(magrittr)  
 library(tidyverse)  
 library(tidylog)  
 # for handling remote-sensing data  
 library(satellite)  
  
 # INPUT & OUTPUT PATHS -----------------------------------------------------  
  
 input1 <- "some/path/to/input/file"  
 output1 <- "some/path/to/output/file"  
  
 # FUNCTIONS ----------------------------------------------------------------  
   
 some\_unc <- function(data=NULL) {  
 some\_code...  
 }  
  
 # TRANSFORM DATA -----------------------------------------------------------  
  
 # TRAIN MODEL --------------------------------------------------------------  
  
 # FINALLY, SAVE ------------------------------------------------------------

See these resources for example style guides:

* [PEP 8 – Style Guide for Python Code](https://www.python.org/dev/peps/pep-0008/)
* [PEP 257 – Docstring Conventions](https://www.python.org/dev/peps/pep-0257/)
* [Tidyverse Style Guide for R](https://style.tidyverse.org/)
* [Google’s Python Style Guide](http://google.github.io/styleguide/pyguide.html?)
* [Google’s R Style Guide](https://google.github.io/styleguide/Rguide.html)

### Clean Projects

TBD

Example Python Data Science Project, from [Cookiecutter Data Science](http://drivendata.github.io/cookiecutter-data-science/)

├── LICENSE  
├── Makefile <- Makefile with commands like `make data` or `make train`  
├── README.md <- The top-level README for developers  
├── data  
│ ├── external <- Data from third party sources  
│ ├── interim <- Intermediate data that has been transformed  
│ ├── processed <- The final, canonical data sets for modeling  
│ └── raw <- The original, immutable data dump  
│  
├── docs <- Code documentation  
│  
├── models <- Trained & serialized models, model predictions or summaries  
│  
├── notebooks <- Jupyter notebooks  
│  
├── references <- Data dictionaries, manuals, and all other explanatory materials  
│  
├── reports <- Generated analysis as HTML, PDF, LaTeX, etc.  
│ └── figures <- Generated graphics & figures used in reporting  
│  
├── requirements.txt <- Requirements file for reproducing the analysis environment, e.g.  
│ generated with `pip freeze > requirements.txt`  
│  
└── src <- Source code for use in this project  
 ├── \_\_init\_\_.py <- Makes src a Python module  
 │  
 ├── data <- Scripts to download or generate data  
 │ └── make\_dataset.py  
 │  
 ├── features <- Scripts to turn raw data into features for modeling  
 │ └── build\_features.py  
 │  
 ├── models <- Scripts to train models & to make predictions   
 │ ├── predict\_model.py  
 │ └── train\_model.py  
 │  
 └── visualization <- Scripts to create visualizations  
 └── visualize.py

### Clean Transitions

TBD

## Virtual Environment Management

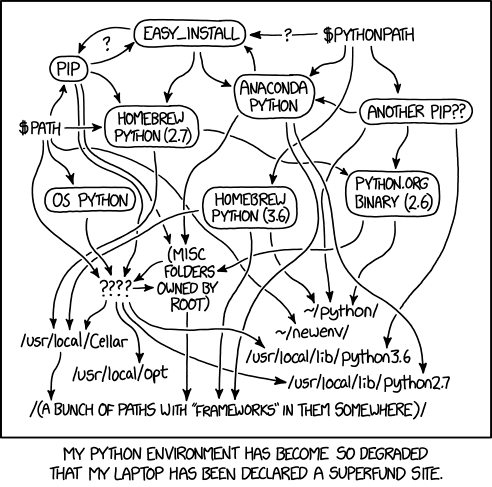
Virtual environments let you have a stable, reproducible, and portable environment. You are in control of which packages versions are installed and when they are upgraded.

Great data science work should be **reproducible**. Being able to repeat experiments is the foundation of all science. Reproducing work is also critical for business applications: scheduled reporting, team collaboration, project validation. Reproducibility is a best practice in data science as well as in scientific research, and in many ways, comes down to having a software engineering mentality. Its core achievement is setting up all of the processes in a way that is repeatable (preferably by a computer) and well documented. The primary and first step in creating reproducible data science products is establishing virtual environments. Virtual environments allow scientist to share work, and most importantly, for team members or other scientist to be able to replicate their work. Here are a few signs to suggest environment management has gone wrong:

* Code that used to run no longer runs, even though the code has not changed.
* You are afraid to upgrade or install a new package, because it might break your code or someone else’s.
* Typing install.packages (in R) in your environment doesn’t do anything, or doesn’t do the right thing.

Another benefit of environment management is **portability**. This means the libraries, packages, and dependencies for one project/task can move from one team member’s machine to the next (assuming the same OS architecture). This is great for project transitions, as new teams will not need to download and install all the required dependencies onto their machines in order to execute projects.

### Python Environment Management



xkcd comic, python environment

Hands-down the most popular environment management software for Data Scientists is Anaconda, or it’s primary core software, conda. (Ana)conda is an environment management system built specifically to provide utilities for data science. Anaconda is a free and open-source distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.), that aims to simplify package management and deployment. Here’s a short list of benefits to using Anaconda or conda:

* Providing prebuilt packages which avoid the need to deal with compilers or figuring out how to set up a specific tool.
* Managing one-step installation of tools that are more challenging to install (such as TensorFlow or IRAF).
* Allowing you to provide your environment to other people across different platforms, which supports the reproducibility of research workflows.
* Allowing the use of other package management tools, such as pip, inside conda environments where a library or tools are not already packaged for conda.
* Providing commonly used data science libraries and tools, such as R, NumPy, SciPy, and TensorFlow. These are built using optimized, hardware-specific libraries (such as Intel’s MKL or NVIDIA’s CUDA) which speed up performance without code changes

Here’s an example using the conda command prompt to setup a project directory, create a conda environment, and initializing a git repository:

mkdir new\_project  
cd new\_project  
conda create --prefix env python=3 # create new directory `env` in current directory  
conda activate ./env  
(/Users/<you>/../new\_project/env) conda install --channel conda-forge jupyterlab matplotlib  
(/Users/<you>/../new\_project/env) pip install hdbscan  
(/Users/<you>/../new\_project/env) conda deactivate  
git init  
echo "env/\*" > .gitignore  
git add .  
git commit -m "Initial commit"

Anaconda python environment management. Details found here: <https://www.anaconda.com/>

Python also includes two options in the standard library for creating virtual environments:

* venv is available by default in Python 3.3 and later, and installs pip and setuptools into created virtual environments in Python 3.4 and later.
* virtualenv needs to be installed separately, but supports Python 2.7+ and Python 3.3+, and pip, setuptools and wheel are always installed into created virtual environments by default (regardless of Python version).

More details on python virtual environments found here <https://packaging.python.org/tutorials/installing-packages/#creating-virtual-environments>

### R Environment Management

TODO: Expand section

* [Packrat](https://rstudio.github.io/packrat/) (deprecated soon)
* [renv](https://blog.rstudio.com/2019/11/06/renv-project-environments-for-r/)
* [RStudio Environments](%3Chttps://environments.rstudio.com/)

## Test Driven Development

Test Driven Development (TDD) is the practice of writing a failed test case (unit testing) and then writing the production code to make the test case pass. Finally, a check for any code refactoring opportunities is made. TDD is the “Red, Green, Refactor” system based on the working model proposed by Kent Beck. An updated version of this system that incorporates modern version control is “Red, Green, Refactor, Commit”

Test Driven Development: By Example

— Kent Beck, 2003

The longer explanation: TDD is an evolutionary approach (relying on incremental improvements) to software development. TDD goes along well with agile processes, but also mimics the scientific method via processes of hypothesizing, testing, and theorizing. More specifically, TDD can be considered a subset of the scientific method: making a logical proposition of validity, sharing results through documentation, and working in feedback loops. Most machine learning practitioners apply some form of the scientific method, and TDD forces you to write cleaner and more stable code while doing so. TDD takes 15–35% more time in active development mode, but it also has the ability to reduce bugs up to 90%. Another clear reason to use TDD is for the benefit of documenting how the code is intended to work. As code becomes more complex, the need for a specification increases—especially as people are making bigger decisions based on what comes out of the analysis.

Thoughtful Machine Learning with Python: A Test-driven Approach

— Mathew Kirk, 2014

See these for more details:

<http://engineering.pivotal.io/post/test-driven-development-for-data-science/>

<http://agiledata.org/essays/tdd.html>

### Unit Testing Frameworks

TODO:

* Add paragraph on basics of unit testing

### R Unit Testing

* [testthat](https://testthat.r-lib.org/)

### Python Unit Testing

* [real python, general testing in python](https://realpython.com/python-testing/)
* [unittest, python’s standard testing library](https://docs.python.org/3/library/unittest.html)
* [pytest, python’s non-standard testing library](https://docs.pytest.org/en/latest/index.html)

# DevOps

## Why it matters: TLDR

DevOps is the combination of cultural philosophies, practices, and tools that increases an organization’s ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes. This speed enables organizations to better serve their customers and compete more effectively in the market. In short, why DevOps matter:

* Reduce Human Error
* Reduce labor intensive build and test tasks so development teams can focus on business function delivery
* Provide structure and consistency to development and operations tasks

The following are DevOps best practices: 1. Continuous Integration 2. Continuous Delivery 3. Microservices 4. Infrastructure as Code 5. Monitoring and Logging 6. Communication and Collaboration

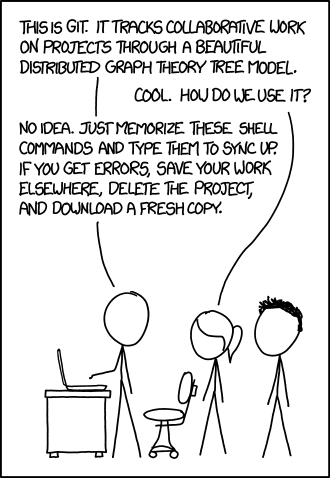
## DevOps Concepts

### Continuous Integration

Continuous integration is a DevOps software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration most often refers to the build or integration stage of the software release process and entails both an automation component (e.g. a CI or build service) and a cultural component (e.g. learning to integrate frequently). The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

With continuous integration, developers frequently commit to a shared repository using a version control system such as Git. Prior to each commit, developers may choose to run local unit tests on their code as an extra verification layer before integrating. A continuous integration service automatically builds and runs unit tests on the new code changes to immediately surface any errors.

### Continuous Delivery



xkcd comic, git

Continuous delivery is a software development practice where code changes are automatically prepared for a release to production. A pillar of modern application development, continuous delivery expands upon continuous integration by deploying all code changes to a testing environment and/or a production environment after the build stage. When properly implemented, developers will always have a deployment-ready build artifact that has passed through a standardized test process.

Continuous delivery lets developers automate testing beyond just unit tests so they can verify application updates across multiple dimensions before deploying to customers. These tests may include UI testing, load testing, integration testing, API reliability testing, etc. This helps developers more thoroughly validate updates and preemptively discover issues. With the cloud, it is easy and cost-effective to automate the creation and replication of multiple environments for testing, which was previously difficult to do on-premises.

CI/CD tools can help a team automate their development, deployment, and testing. Some tools specifically handle the integration (CI) side, some manage development and deployment (CD), while others specialize in continuous testing or related functions.One of the best known open source tools for CI/CD is the automation server Jenkins. Jenkins is designed to handle anything from a simple CI server to a complete CD hub.

Below is a list of common DevOps tools:

* Git(Hub)
* Jira
* Jenkins
  + [Deploying Jenkins on OpenShift/AWS](https://www.openshift.com/blog/deploying-jenkins-on-openshift-part-1)
* Tekton
  + [Deploying Tekton on OpenShift/AWS](https://www.openshift.com/learn/topics/pipelines)
  + Tekton Pipelines is a CI/CD framework for Kubernetes platforms that provides a standard cloud-native CI/CD experience with container
* Docker
* Kubernetes

### Testing, Staging, & Production Environments

TODO:

* Expand section
* Dev Testing (local dev’s machine)
* Local Testing (human testing on separate local machine)
* Production Testing (AWS)
* Staging Environments
* Deploy to Production (AWS)

### APIs, Microservices, and Containers

The microservices architecture is a design approach to build a single application as a set of small services. Each service runs in its own process and communicates with other services through a well-defined interface using a lightweight mechanism, typically an HTTP-based application programming interface (API). Microservices are built around business capabilities; each service is scoped to a single purpose. You can use different frameworks or programming languages to write microservices and deploy them independently, as a single service, or as a group of services.

**APIs & Microservices**

TODO: Expand section with details on APIs and microservices

**Docker Containers**

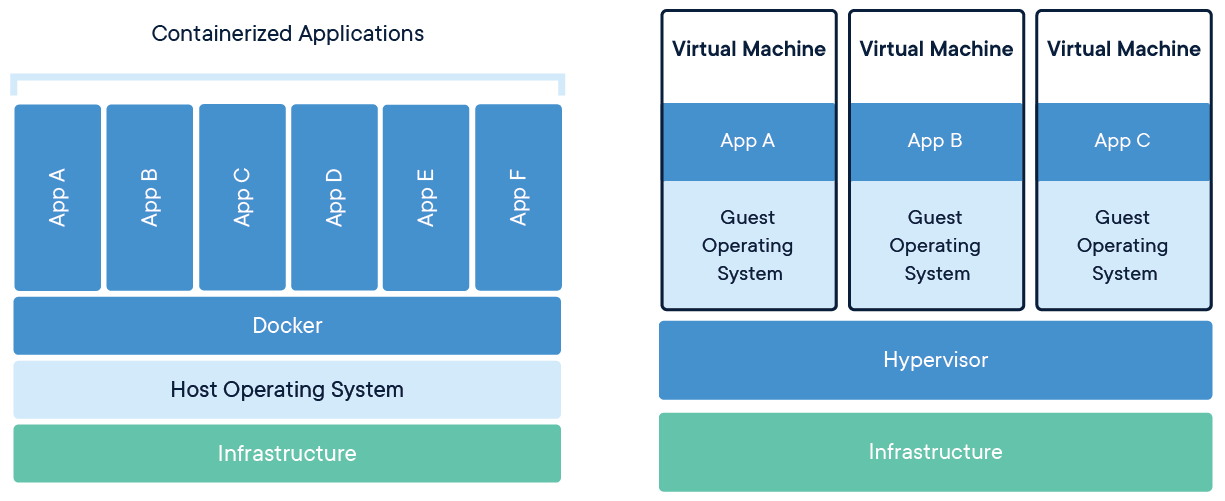
Containers are a standardized unit of software that packages code and all its dependencies so the application runs quickly and reliably from one computing environment to another, solving the “it works on my machine” headache. A container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Docker is the de facto standard to build and share containerized applications. Docker, as a containerization tool, is often compared to virtual machines (VM). Containers and VMs have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient, and container images demand far fewer resources to run than VMs.

Containerized applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a analyst’s personal laptop. Containers create consistent environments, which results in analysts and developers spending less time debugging and diagnosing differences in environments, and more time building models, running analysis, and shipping new functionality and insights.

Containers are perfectly suited for creating microservices. Microservices divide out “monolithic” applications into separate components, allowing the different parts application to be scaled, modified, and/or serviced separately. Lightweight, portable, and self-contained, Docker containers make it easier to build microservices.

Docker containers provide the following:

* Docker is the industry standard for containers, so they could be portable anywhere
* Containers enable more efficient use of system resources, faster software delivery, and application portability
* Containers provide consistent environments, which translates to productivity
* Containers are lightweight and efficient
* Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry



Docker Containers vs Virtual Machines

**Example: Turning an AI model into Production**

A very simplified, python centric, Machine Learning DevOps workflow that utilizes ML models, APIs/microservices, and docker containers is presented below. It is important to note the below example does not include many of the standard DevOps topics and practices (i.e. version control, development/production environments, or CI/CD).

* Transform an existing machine learning algorithm into an API, operating as a microservice: This process (model to API to microservice) creates a service you can “ask questions”. For example. if you created a machine learning model that predicts house prices based on square footage & made it an API you could send your flask app an HTTP request saying “given this square footage what would be the price?” & it would return an answer.
  + Flask, a Python framework for building web services (think APIs, websites, etc.), can be used to load a model trained in sklearn and serve predictions to an endpoint. Gunicorn (pronounced jee-unicorn) provides an HTTP server. Flask comes with a small server for running locally during development, but it’s not recommended for production, which is why gunicorn is recommended.
* Use Docker to containerize the machine learning model flask/gunicorn app: Docker is a service that “containerizes” the flask app into an image. This image has everything you need to run an app bundled together (like a virtual machine), so it’s guaranteed to run on any machine that has Docker.
* Deploy the container image into production using Kubernetes: Kubernetes can be used to deploy the docker image on a network. Once you make changes to the flask app and build a new image, you can deploy the new image into production. If Kubernetes is available, then docker containers is recommended. However, if Kubernetes is not available then deploying the flash/dash/RShiney apps on a VM is the next best option.

## DevOps Principles

### Infrastructure as Code

Infrastructure as code is a practice in which infrastructure is provisioned and managed using code and software development techniques, such as version control and continuous integration. The cloud’s API-driven model enables developers and system administrators to interact with infrastructure programmatically, and at scale, instead of needing to manually set up and configure resources. Thus, engineers can interface with infrastructure using code-based tools and treat infrastructure in a manner similar to how they treat application code. Because they are defined by code, infrastructure and servers can quickly be deployed using standardized patterns, updated with the latest patches and versions, or duplicated in repeatable ways.

### Configuration Management

Developers and system administrators use code to automate operating system and host configuration, operational tasks, and more. The use of code makes configuration changes repeatable and standardized. It frees developers and systems administrators from manually configuring operating systems, system applications, or server software.

### Policy as Code

With infrastructure and its configuration codified with the cloud, organizations can monitor and enforce compliance dynamically and at scale. Infrastructure that is described by code can thus be tracked, validated, and reconfigured in an automated way. This makes it easier for organizations to govern changes over resources and ensure that security measures are properly enforced in a distributed manner (e.g. information security or compliance with PCI-DSS or HIPAA). This allows teams within an organization to move at higher velocity since non-compliant resources can be automatically flagged for further investigation or even automatically brought back into compliance.

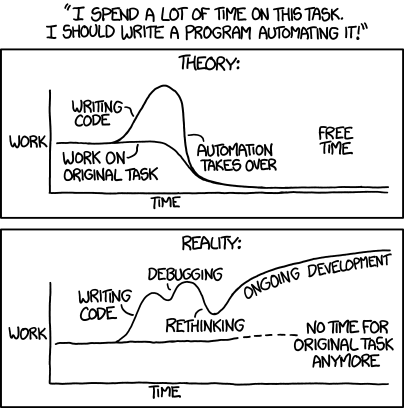
### Monitoring and Logging

Organizations monitor metrics and logs to see how application and infrastructure performance impacts the experience of their product’s end user. By capturing, categorizing, and then analyzing data and logs generated by applications and infrastructure, organizations understand how changes or updates impact users, shedding insights into the root causes of problems or unexpected changes. Active monitoring becomes increasingly important as services must be available 24/7 and as application and infrastructure update frequency increases. Creating alerts or performing real-time analysis of this data also helps organizations more proactively monitor their services.

### Communication and Collaboration

Increased communication and collaboration in an organization is one of the key cultural aspects of DevOps. The use of DevOps tooling and automation of the software delivery process establishes collaboration by physically bringing together the workflows and responsibilities of development and operations. Building on top of that, these teams set strong cultural norms around information sharing and facilitating communication through the use of chat applications, issue or project tracking systems, and wikis. This helps speed up communication across developers, operations, and even other teams like marketing or sales, allowing all parts of the organization to align more closely on goals and projects.

### Automation



xkcd comic, automation

TODO:

* Expand sections

The concept of automated build refers just to automated software builds but also to automated provisioning and post-provisioning of infrastructure. A variety of tools including Maven, Chef, Terraform and Jenkins participate in the automated build processes.

Automation provides the following benefits: - Reduced errors - Increased efficiency - Improved quality

## DevOps Roadmap

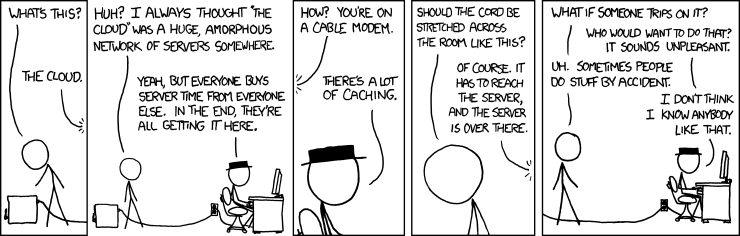
### Short-term

* Expand section

### Long-term

* Data Migration
* Data Updates
* Data/Model Quality Control
* CI/CD
* End-to-end Pipelines

## Cloud Migration: AWS



xkcd comic, the cloud

TODO:

* Expand sections
* Computing Power: Clusters & Processing
* Data Storage: Big Data Beats Smart Algorithms
* Cost-effective
* Scalability

### AWS, EC2, S3

* AWS: Cloud provider
* EC2: Virtual machine(s), instances, clusters
* S3: Data Storage (“Data Lakes”, “Data Warehouse”)

### Overall Migration Strategies

1. Portfolio Discovery and Planning

* This is the stage where basic questions are asked about the overall department(s) portfolio of IT infrastructure/applications and the answers are recorded and utilized during the next steps in migration process. Questions include:
  + What’s in your environment?
  + What are the interdependencies?
  + What will you migrate first and how will you migrate it?

1. Designing, Migrating, and Validating Applications

* Here the focus moves from the portfolio level to the individual application level. Each application is designed, migrated, and validated using one of six strategies listed in the following section.
* To scale the migration quickly, agile teams can be created to focus on different types of “migration themes” (e.g. data, models, dashboards). This is also a good time to formulate a strategy for testing and decommissioning old system(s).

1. Modern Operating Model

* As applications are migrated, you iterate on your new foundation, turn off old systems, and constantly iterate toward a modern operating model.

**Coordination & close partnership with the Cloud Team will be vital during most steps outlined in these strategies**

### Application Migration Strategies

Pro Tip: start with the simplest application/workflow, then build skills and technical knowledge to aid in the migration of larger and more complex applications

1. Rehosting — Otherwise known as “lift-and-shift”

* This is what it sounds like: transfer as much of the existing application to the cloud as-is. Most rehosting can be automated with tools (e.g. AWS VM Import/Export, Racemi), although some customers prefer to do this manually as they learn how to apply their legacy systems to the new cloud platform. Applications are easier to optimize/re-architect once they’re already running in the cloud.

1. Replatforming — Otherwise known as “lift-tinker-and-shift”

* This strategy includes making a few cloud (or other) optimizations in order to achieve some tangible benefit, but there aren’t changes the core architecture of the application. Examples of cloud optimizations include reducing the amount of time managing database instances by migrating to a database-as-a-service platform like Amazon Relational Database Service (Amazon RDS), or migrating your application to a fully managed platform like Amazon Elastic Beanstalk.

1. Repurchasing — Moving to a different product

* This strategy is most commonly used as a move to a SaaS platform. Examples include moving a CRM to Salesforce.com, an HR system to Workday, a CMS to Drupal, and so on.

1. Refactoring / Re-architecting

* Re-imagining strategy involves considering how the application/model/product is architected and developed, typically using cloud-native features. This is typically driven by a strong business need to add features, scale, or performance that would otherwise be difficult to achieve in the application’s existing environment.

1. Retire — Get rid of

* This strategy identifies which applications simply need to be dropped. Once all applications and/or products have been discovered in your existing environment, you might ask each functional area who owns each application. Sometimes it’s discovered that certain applications/models/products of an enterprise IT portfolio is no longer useful, and can simply be turned off. These savings can boost the business case, direct your team’s scarce attention to the things that people use, and lessen the surface area you have to secure.

1. Retain — Otherwise known as “revisit”

* This strategy is the do-nothing for now strategy. Sometimes applications are going through depreciation, or conversely, were recently upgraded, or are otherwise not suitable to migrate. You should only migrate what makes sense; and, as the gravity of your portfolio changes from on-premises to the cloud, you’ll probably have fewer reasons to retain.

**Coordination & close partnership with the Cloud Team will be vital during most steps outlined in these strategies**

### Short-term tasks

The short-term goal for migrating to the cloud is to configure appropriate cloud (user) settings, develop a baseline cloud environment, stand up a simple proof-of-concept program, establish and adopt best practices/workflows, and ideally establish a Cloud Center of Excellence.

* Configuration & Baseline Environment
  + EC2 Instances (virtual machines with enough CPUs)
  + S3 Buckets (load, store, and host data)
  + Configure user permissions
  + API access tokens, requires
    - UNIX attributes
    - AWS console and CLI (this is where you are given an access token)
    - for multi-user shell access, see:
      * <https://aws.amazon.com/blogs/developer/cross-account-iam-roles-in-windows-powershell/>
  + Deploy/install baseline required software or docker images
* Develop a simple proof of concept
  + See below outline
* Establish best management practices and workflows for users
  + Starting and stopping AWS instances
  + Checking processes & status
  + Establish standard connections to data and instances
  + Outline proper scripting and modeling behaviors

Simple proof-of-concept outline

1. Create a virtual machine and deploy/install any required software or servers (e.g. RStudio, Anaconda, Chrome Web Browser, Jupyter)
2. Create a virtual environment with the required frameworks, libraries, and packages (e.g. machine learning libraries, tidyverse, ggplot, matplotlib, RShiny, etc.)
3. Generate an AWS access token to connect local machines to AWS cloud service, which enables shells scripts to be used to manage and automate common AWS workflows via the AWS CLI.
4. Using the AWS CLI command line and shell scripts, automatically start/ssh/check/stop EC2 instances. For example, write one shell script that includes the multiple lines of AWS commands below, save it in a scripts folder, and then execute the script from the AWC CLI terminal by simply typing scripts/connect\_to\_instance.sh.

aws ec2 describe-instance-status --instance-ids <your-instance-id>  
aws ec2 start-instances --instance-ids <your-instance-id> # Start instance  
aws ec2 stop-instances --instance-ids <your-instance-id> # Stop instance

1. Upload multiple csv file(s) extracts on an S3 bucket, stored within an input\_data folder directory
2. Upload script(s) onto the EC2 instance, stored in a scripts folder. The script(s) should read-in above csv file in the input\_data folder
3. Using a AWS CLI shell script, start instance/virtual machine that has RStudio installed
4. Run the script using an installed application (e.g. RStudio, VS Code) that reads the hosted WRD csv file(s) and outputs a new csv file into an output\_data directory, all within the EC2 virtual machine

### Mid/Long-term tasks

Establish an end-to-end data science pipeline

* Migrate most/all datasets to S3 bucket
  + Automatic migration & transfer to (a database) S3 bucket
* Monitoring data logistics
  + data transfers
  + data quality
  + data (pre)processing
  + etc.
* Establish/Deploy docker images for Data Scientist (or install manually, as appropriate)
  + R, Python, Rstudio, Jupyter Labs, etc.
  + Existing docker images: ArcGIS Notebooks, ArcGIS Insights
  + For docker images, see:
    - <https://aws.amazon.com/getting-started/hands-on/deploy-docker-containers/>
    - <https://hub.docker.com/r/jupyter/datascience-notebook/>
    - <https://jupyter-docker-stacks.readthedocs.io/en/latest/index.html>
    - <https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html>
* Fully migrate and host appropriate/feasible analytic and data science solutions
  + Weekly and Quarterly Reporting
  + AI models
  + Dashboards
  + etc.
* Automatic logging, reporting, and CI/CD (Git, Jenkins, etc.)
* Establish testing, staging, and production environments
  + Three different virtual machines: one for development, testing, and production environments
  + Two separate S3 buckets: one for development and testing and one for production

# R Markdown Reference

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com> and <https://bookdown.org/yihui/rmarkdown/>

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document.