Mid-level data engineer theory course

Luca Mircea

Premise

There is nowhere an agreed upon standard that defines what makes a mid-level data engineer, and part of why this is so is because the role is very varied and diverse, and nobody is expected to know everything, especially early on in their career. Nonetheless, there are some topics that do appear in job postings more often than not, and foundational skills that can help in any position.

The course at hand is meant to serve as an introduction for some of the more relevant of these concepts and show how they work together as a way to help students get a solid grasp of them.

The choice of curriculum is based on the author's personal experience with interviews and technical assignments given for mid-level data engineering roles and is by no means extensive. Nonetheless, the material contained in this course seems to be enough to help somebody find a relevant position, and one could argue that this is what it means to become a data engineer.

This course does not even aim to cover everything: it is also very important to have strong SQL skills (something that deserves its own course) and decent cloud knowledge (such as can be obtained by preparing for a certification exam). In order to truly be prepared for a mid-level data engineering role, students are also advised to develop in those areas.

As the author of this course, I hope you find it useful, and I wish you the best of luck with your education and career!

Course outline

The idea of this course is to learn how to build, package, and deploy an app, a skill that is universally needed in software engineering. This is because code doesn't do anything unless it's put into production, and any engineer worth their salt needs to be able to put their own code into production.

To cover this entire process, the course will begin with the code itself, since no app can run without it, and it is the basis of everything else.

The next step is going to be to look into unit testing, which is basically additional code that ensures the main code works as expected – this may seem silly as a concept, but is a very

powerful one in the arsenal of many, so much so that “test-driven development” is an entire paradigm.

After the code is written and tested, it becomes necessary to ensure that it can be re-built and also run on other machines; this is what dependency management does, by helping create and keep the “recipe” for building the program at the code level.

This flows elegantly into the subsequent topic, which is known as containerization, or even dockerization since Docker is the most common app used for this, which works together with the “recipe” created in the previous step to recreate a “clean slate” sort of environment on any machine where the code would run. Docker/containerization is the backbone of any piece of modern engineering, since it’s part of what allows for distributed computing, so it is crucial and fundamental to master this topic.

When one has a containerized piece of code, the only missing piece is to deploy it to production – this is where CI/CD comes in as an automated process that handles this for you so that you don’t have to do it manually every time. In most companies, this CI/CD flows are offered “as a service” to developers by the DevOps engineer, whose job it is to make it easy for everybody else to deploy their code. Nonetheless, to make use of such a flow, one needs to understand what it consists of, and this is why it’s covered in this roadmap.

Finally, when one knows all the steps outlined above, one is ready to build an app! To take advantage of this new skill and to make sure it sticks with the students, the last part of this course is a project where all the pieces come together into one deliverable that would impress any hiring manager looking for a mid-level engineer.

In summary, students will go step-by-step through the app development process, from writing the code to delivering the final app. This is helpful for learning sought-after skills, but also because it gives a better grasp of how all software works. Whoever completes this course, has decent SQL knowledge, and a cloud certification should have everything it takes to find a job as a mid-level data engineer.

This course is meant, at least for now, as more of an outline for self-study, hands-on practice, and question sessions. Therefore, it will consist of weekly sessions where the topic of the previous week is discussed along with any unanswered questions the students might have. They then have time to study on their own the subject of the next stage, to collect information, and to write a short report/essay about it (minimum 900 words), touching upon key points that are stated in the curriculum. The students are encouraged to write and keep these essays in such a way that it’s easy to use them for revising the content before an interview or even during an actual job. To ensure proper development, each stage also comes with a hands-on assignment related to the topic covered, which also serves as a preparation for the final task. Then, once

stages 1-5 have been covered, what's left is putting everything together in one big project, so the students will have twice the time as between two sessions to build as nice an app as possible, using everything they will have learnt by that time in the course.

Good luck!

Stage 1 – Python as a coding language

Theory

Python is the coding language most often associated with data engineering, and also with numerous other use cases. However, like everything else, it has upsides and downsides, as well as particularities in how it works. The aim of this first session is to familiarize the students with Python as a tool in terms of what it does well and what it doesn't, but also with how to get it running. Therefore, the essay/report should cover the following points:

- Pros and cons of **Python** – why is it commonly used? When to use? When not to use? And how does it compare with other languages like C#, JavaScript, or Scala?
- **Virtual environments** – what they are, why we use them, how to create one and work with it. Bonus: learning how to do it with multiple versions of Python (e.g. 3.10 and 3.11)
- Modularization and **init.py** – learning how to split one app into multiple files (stored in a folder called `src`) and the role of the init.py file in this
- **Entry points** – what are they, what are they used for, how to use them
- **Environment files** – what they are, what they are used for. Look into `.env` and the Python package called `dotenv`
- Bonus: semi-advanced concepts for the code itself, such as **list comprehension**, **typing** (in functions, basically indicating the type of objects going into and out of a function), **decorators** (use retry from tenacity, write your own), **Exceptions** (how to raise your own custom errors), **try-except blocks** (how to manage the failures of your code more gracefully and in ways that help you debug it)

Practice

Note: it is recommended you start with a clean repo from scratch and put everything on GitHub from the beginning – eventually this becomes a big part of the entire project, so you might as well save yourself the hassle of transferring later.

Please write Python code to **retrieve data from an API** of your choice (ideas: weather, cryptocurrency, sports, transportation, holidays, etc. – you can be creative, the internet is full of

free-to-use APIs). The **data should be saved locally** in a folder parallel to `src`, named `data`, and also **saved into an in-memory database** (basically a locally run, mock database used to test code against a DB). Once the data is "uploaded" to the database, the program **should print a message saying "Data uploaded successfully"**. The code should be **split into four separate** `.py` files: `constants.py` (where you store API settings and retrieve the API key as a constant from the `.env` file – this might be a bit difficult, but you can figure it out with ChatGPT), `extract_data.py` (where you store the function for calling the API, it must retrieve the data and return it as a pandas.DataFrame object, and the code should indicate this), `upload_to_db.py` (where you store the code that creates the mock DB and adds to it the data in `data`), and `main.py` (which puts together all the code from the other files into one neat `main()` function). Watch out: importing from one .py file into another might not work if you don't have init.py setup properly. Please **try implementing at least three of the bonus points mentioned above** in your code, ideally all 5.

Your folder structure should look something like this:

```
|_data
  |_your_data.csv
|_src
  |_main.py
  |_constants.py
  |_extract_data.py
  |_upload_to_db.py
  |_.env
  |_init.py
|_venv
```

Stage 2 – Unit testing

Theory

Unit tests are basically pieces of code that confirm that your code works as expected. This can be particularly useful when you're implementing some finnicky logic and you want to ensure it executes strange edge cases accordingly, but can be used for so many other purposes also. In fact, it's such a big topic that "Test-Driven Development" is an entire paradigm in software that many swear by. They are more important in other parts of software engineering, but can be

very useful in data engineering as well, e.g. to ensure that your data processing functions output dataframes with the right format no matter the data fed into it. The essay regarding this topic should include:

- What is unit testing, how is it done, and what is it used for?
- What kinds of tests are there?
- What are testing best practices? What about testing don'ts?
- Why is Test-Driven Development such a big paradigm in software development?

Practice

Implement three tests to your code from stage 1 using `pytest`. Make sure that at least one of these tests checks the output of your API extraction function is a pandas.DataFrame (hint: you don't need to query the actual API as part of the test, for that you can use a *mock*). If possible, create a schema for your data and write a test for a function to ensure that `string` data appears in `string` columns, `integer` for `integer`, etc.

Stage 3 – Packaging and dependency management

Theory

As you've probably seen by now, barebones Python doesn't go very far by itself, nor does it make much sense. Therefore, you probably understand why managing dependencies is important, on an intuitive level, but in this stage we will formalize this knowledge. The key points to be covered in the essay about packaging and dependency management are:

- Why is this done?
- Managing dependencies via requirements.txt – the easiest way to do it.
- Pipenv, setuptools, poetry – advanced tools for dependency management. Please explain some of the differences between them, come up with 1-2 reasons why using one of these might make more sense than just requirements.txt (if you're going to use ChatGPT for the reasons, then please write three)

Practice

Prepare a requirements.txt file, as well as a project file (such as Pipenv, pyproject.toml, etc.) for your code written in stage 1. Please make sure that pytest (required during stage 2) is also installed during setup. Delete your virtual environment folder from the project and re-build it from scratch using requirements.txt, and also your preferred setup tool (whether it's Pipenv or

pyproject.toml). Bonus points: add your code to a GitHub repo, delete it from local, clone the repo locally, then set it up using your preferred setup tool.

Stage 4 – Containerization with Docker

Theory

Every computer is a little bit different due to slight changes in versions, etc., or even OS. This can be a massive hurdle for making software work, and this is the problem that Docker seeks to address. Since it creates a "clean slate" environment for code, it is crucial for modern programming when we need to run the same logic on multiple machines and cloud servers too. Aside from the replicability of the environment, Docker also buys us modularization through the creation of containers that work together, and this is the backbone of modern-day distributed systems. The importance of Docker cannot be understated, it's probably the most important part of this entire course, and it's something that's also valid for proper software engineering. Therefore, the points that should be covered while studying Docker are:

- Why is this a thing? How does it work?
- Why is Docker important for distributed systems? How does it interact with Kubernetes?

Practice

Practice is more important than theory with Docker. Please dockerize the code you wrote in stages 1 to 3: write a Dockerfile that can be run which compiles and runs the Python script in `main()`. You will need to install Python and your dependencies in the container – please do so both via the traditional `requirements.txt` way, and by using `Pipenv` or `pyproject.toml`. You should make is possible to run `docker run container-name` and it should print out the **"Data uploaded successfully"** message coded in stage 1 after it uploads data to the mock database.

Stage 5 – CI/CD with GitHub actions

Theory

When the code is ready, tested, reproducible, it is time to deploy it to prod – this is what the CI/CD flow does in short, but there is a bit more depth to this process, especially since it's quite abstract. The points covered by the investigation should be:

- What is CI and why do we do it? Please give three examples of checks that could be run as part of this process

- What is CD and why do we do it? Please give three examples of steps that could be executed as part of this process
- What is the idea behind CI/CD?
- What are some common tools/ways of doing it?

Practice

Add your code from steps 1 to 4 to Github (if you haven't done so already), build 3 to 4 CI/CD checks: two mandatory steps would be to add a linter (such as black or ruff) and a script for running your tests automatically. The third check should trigger an action, like e.g. print "The code is being deployed" – this would serve as a mock deployment, since we have no actual prod to deploy to as part of this project.

Stage 6 – Integration: database with StreamLit

Theory

All the theory has been covered, so now the best way to let it sink in is to put everything into practice into one big project: creating a queryable interface with StreamLit. The idea is to pull data from an API and to upload it to a mock, in-memory database that can be queried via a UI using a StreamLit front-end. The only theoretical part of this stage is to write nice documentation for the project, including a neat `README.md` file.

Practice

Please build your code/repo from stages 1 to 5 into a dockerized StreamLit app that opens in the browser in `localhost:8080` or `http://0.0.0.0:8080` and shows a UI where users can input SQL queries. The queries should actually run against the data you retrieved from the API and loaded into the database, and they should return either results or error messages. When the first query runs against the API data in this environment, you can consider the course complete – congratulations! You now have the theoretical foundations to be a mid-level data engineer!