Chapter 11. Building MLOps Command Line Tools and Microservices

By Alfredo Deza

> *The Japanese bombed Pearl Harbor on December 7, 1941 and it at once became impossible to buy tires, clearly needed for the long trip to California. My father scoured the country for old tires for the Ford and for the small house trailer that we had used for several shorter family trips. We left Cincinnati in February 1942 with 22 tires strapped to the roofs of the trailer and the car and before we got to California we used them all.*
>
> —Dr. Joseph Bogen

Building command line tools is how I got started with Python years ago, and I believe it is a perfect intersection between software development and machine learning. I remember all those years ago when I struggled to learn new Python concepts that felt very foreign to me as a Systems Administrator: functions, classes, logging, and testing. As a Systems Administrator, I was mostly exposed to shell scripting with Bash writing top-to-bottom instructions to get something done.

There are several difficulties with trying to tackle problems with shell scripting. Handling errors with straightforward reporting, logging, and debugging are all features that don't take much effort in other languages like Python. Several other languages offer similar features like Go and Rust and should hint why using something other than a command language like Bash is a good idea. I tend to recommend using a shell scripting language when a few lines can get the task done. For example, this shell function copies my public SSH key into a remote machine into the authorized keys, allowing my account to access that remote server without requiring a password:

```
ssh-copy-key() {
  if [ $2 ]; then
      key=$2
  else
      key="$HOME/.ssh/id_rsa.pub"
  fi
  cat "$key" | ssh ${1} \
      'umask 0077; mkdir -p .ssh; cat >> .ssh/authorized_keys'
}
```

NOTE

The sample Bash function will probably not work well in your enviroment. If you want to try it out, the paths and configuration files will have to match.

Doing the same with a language like Python wouldn't make much sense because it would take many more lines of code and probably a few extra dependencies installed. This function is simple, only does one thing, and is very portable. My recommendation is to look beyond a shell scripting language when the solution is more than a dozen lines or so.

TIP

If you find yourself creating small snippets of shell commands often, it is good to create a repository to collect them. The repository for my aliases, functions, and configurations should show you a good way to start if you need a reference.

The way I learned Python was by automating tedious, repetitive tasks with command line tools, from creating client websites using templates to adding and removing users from the corporate servers. My recommendation to get engaged in learning is to find an interesting problem with a direct benefit for yourself. This way of learning is entirely different from how we were taught in school, and it doesn't necessarily apply to any situation, but it is a good fit for this chapter.

Depending on how you create command line tools, they can be straightforward to install. As the example of a shell command that should work

in any Unix environment with ease, there is a similarity with containers and microservices. Because the container bundles the dependencies together, it will work in any system with a proper runtime enabled. We've already gone through some of the components of microservices in "Containers", describing some critical differences from monolithic applications.

But there is more to microservices than containers, and almost all cloud providers offer a *serverless* solution. Serverless allows a developer to concentrate on writing small applications without worrying about the underlying operating system, its dependencies, or the runtime. Although the offering may appear overly simplistic, you can leverage the solution to create whole HTTP APIs or a pipeline-like workflow. You can tie all of these components and technologies back to the command line and some ML features from cloud providers. The mix-and-match aspect of these technologies means that an engineer can craft creative solutions to tricky problems with very little code. Whenever you can automate tasks and enhance productivity, you are applying solid operational skills to help you get models into production robustly.

Python Packaging

Many useful Python command line tools start as a single script file, and then they tend to grow to more complex scenarios with other files and perhaps dependencies. It isn't until the script needs extra libraries that it is no longer feasible to keep the script without packaging. Python packaging is not very good. It pains me to say that after well over a decade of Python experience, but packaging is still an aspect of the language ecosystem full of tricky (unresolved) problems.

If you are tinkering and trying out some automation *with no external dependencies*, then a single Python script is fine without packaging. If, on the other hand, your script requires some other dependencies and perhaps consists of more than one file, then you undoubtedly should consider packaging. Another useful feature of a properly packaged Python application is that it can be published to the Python Package Index so that others can install it with tools like *pip* (the Package Installer for Python).

A few years ago, it was problematic to install Python packages on a system: it was impossible to remove them. This sounds unbelievable these days, but it was one of the many reasons why "virtual environments" took off. With virtual environments, fixing dependencies was as easy as removing a directory—while keeping the system packages intact. These days, uninstalling Python packages is easier (and possible!), but dependency resolution is still lacking robustness. Virtual environments are then the suggested way to work on Python projects, so environments are fully isolated, and you can resolve dependency problems by creating a new environment.

It shouldn't be surprising to see recommendations throughout this book (and elsewhere in the Python ecosystem) to use the *virtualenv* module. Since Python 3, the common way to create and activate a virtual environment is with the *Python* executable directly:

```
$ python -m venv venv
$ source venv/bin/activate
```

To verify that the virtual environment is activated, the Python executable should now be different from that of the system Python:

```
$ which python
/tmp/venv/bin/python
```

I recommend using proper Python packaging techniques so that you are well prepared when it is needed. As soon as your command line tool needs a dependency, it will be ready to be declared as a requirement. Your tool's consumers will resolve these dependencies as well, making it easier for others to work with your creation.

The Requirements File

As you will see in the next sections of this chapter, there are two popular ways of defining dependencies. One of them is using a *requirements.txt* file. Installer tools can use this file like *pip* to get dependencies installed

from a package index. In this file, the dependencies are declared on a separate line, and optionally, with some versions constraint. In this example, the Click framework doesn't have a constraint, and therefore the installer (`pip`) will use the latest version. The Pytest framework is *pinned* to a specific version, so `pip` will always try to find that specific version at install time:

```
# requirements.txt
click
pytest==5.1.0
```

To install dependencies from a *requirements.txt* file, you need to use `pip` :

```
$ pip install -r requirements.txt
```

Although there is no strict naming rule, you can commonly find dependencies in a plain-text file named *requirements.txt*. Project maintainers can define multiple text files with requirements as well. That is more common when development dependencies are different than when shipping to production, for example. As you will see in the next section, there is also a *setup.py* file that can install dependencies. This is a rather unfortunate side-effect of the state of packaging and dependency management in Python. Both files can achieve the goal of installing dependencies for a Python project, but only *setup.py* can package a Python project for distribution. Because a *setup.py* file is executed at install time by Python, it allows doing anything aside from installation tasks. I don't recommend extending *setup.py* to do anything other than packaging tasks to prevent issues when distributing an application.

Some projects prefer to define their dependencies in a *requirements.txt* file and then reuse that file's contents into the *setup.py* file. You can achieve this by reading the *requirements.txt* and using the `dependencies` variable:

```
with open("requirements.txt", "r") as _f:
    dependencies = _f.readlines()
```

Distinguishing between these packaging files and knowing their background is useful to prevent confusion and misuse. You should now feel more comfortable discerning if a project is meant for distribution (*setup.py*) or a service or project that doesn't require installation.

Command Line Tools

One of the Python language features is the ability to quickly create applications with almost anything that you can imagine already included, from sending HTTP requests to processing files and text, all the way to ordering streams of data. The ecosystem of libraries available is vast. It doesn't seem surprising that the scientific community has embraced Python as one of the top languages to tackle workloads that include machine learning.

An excellent way of approaching command line tool development is to identify a particular situation that needs solving. Next time you encounter a somewhat repetitive task, try to build a command line tool to automate the steps to produce the result. Automation is another core principle of DevOps that you should apply whenever possible (and as much as it makes sense) to tasks throughout ML. Although you can create a single Python file and use it as a command line tool, the examples in this section will use proper packaging techniques that will allow you to define required dependencies and get the tool installed with Python installers like `pip`. In the first example tool, I'll show these Python patterns in detail to grasp the ideas behind command line tools that you can apply to the rest of this chapter.

Creating a Dataset Linter

While creating this book, I decided to put together a dataset of wine ratings and descriptions. I couldn't find anything similar, so I started gathering information for the dataset. Once the dataset had a healthy number of entries, the next step was to visualize the information and determine how robust the data was. As usual with the initial data state, this dataset presented several anomalies that took some effort to identify correctly.

One problem was that after loading the data as a Pandas data frame, it was clear that one of the columns was unusable: almost all were `NaN` (also referred to as null entries). Another issue, which might be the worst of them all, was that I loaded the dataset into Azure ML Studio to perform some AutoML tasks that started yielding some surprising results. Although the dataset had six columns, Azure was reporting about forty.

Lastly, *pandas* added unnamed columns when saving the processed data locally, and I wasn't aware of this. The dataset is available to demonstrate the problems. Load the CSV (comma separated value) file as a *pandas* data frame first:

```
import pandas as pd
csv_url = (
  "https://raw.githubusercontent.com/paiml/wine-ratings/main/wine-ratings.csv"
)
# set index_col to 0 to tell pandas that the first column is the index
df = pd.read_csv(csv_url, index_col=0)
df.head(-10)
```

The table output from pandas looks great but hints that one column may be empty:

```
                 name  grape          region     variety  rating  notes
...               ...    ...             ...         ...     ...  ...
32765  Lewis Cella...    NaN  Napa Valley...  White Wine    92.0  Neil Young'.
32766  Lewis Cella...    NaN  Napa Valley...  White Wine    93.0  From the lo.
32767  Lewis Cella...    NaN  Napa Valley...  White Wine    93.0  Think of ou.
32768  Lewis Cella...    NaN  Napa Valley...    Red Wine    92.0  When asked .
32769  Lewis Cella...    NaN  Napa Valley...  White Wine    90.0  The warm, v.

[32770 rows x 6 columns]
```

When describing the dataset, one of the issues is now clear; the *grape* column doesn't have any items in it:

```
In [13]: df.describe()
Out[13]:
       grape        rating
```

```
count    0.0  32780.000000
mean     NaN     91.186608
std      NaN      2.190391
min      NaN     85.000000
25%      NaN     90.000000
50%      NaN     91.000000
75%      NaN     92.000000
max      NaN     99.000000
```

Drop the problematic column, and save the dataset onto a new CSV file, so that you can manipulate the data without having to download the contents every time:

```
df.drop(['grape'], axis=1, inplace=True)
df.to_csv("wine.csv")
```

Rereading the file demonstrates the extra column added by *pandas*. To reproduce the issue, reread the local CSV file, save it as a new file, and look at the first line of the newly created file:

```
df = pd.read_csv('wine.csv')
df.to_csv('wine2.csv')
```

Look at the first line of the *wine2.csv* file to spot the new column:

```
$ head -1 wine2.csv
,Unnamed: 0,name,region,variety,rating,notes
```

The Azure problem was more involved, and it was tough to detect: Azure ML was interpreting new lines and carriage returns in one of the columns as new columns. To find these special characters, I had to configure my editor to show them (usually, they aren't visible). In this example, the carriage return shows as `^M`:

```
"Concentrated aromas of dark stone fruits and toast burst^M
from the glass. Classic Cabernet Sauvignon flavors of^M
black cherries with subtle hints of baking spice dance^M
across the palate, bolstered by fine, round tannins. This^M
```

```
medium bodied wine is soft in mouth feel, yet long on^M
fruit character and finish."^M
```

After dropping the column with no items in it, removing the unnamed columns, and getting rid of carriage returns, the data was now in a much healthier state. Now that I've done my due diligence cleaning up, I want to automate catching these problems. I'll probably forget the deal with the extra columns in Azure or a column with useless values in it a year from now. Let's create a command line tool to ingest a CSV file and produce some warnings.

Create a new directory called *csv-linter* and add a *setup.py* file that looks like this:

```
from setuptools import setup, find_packages

setup(
    name = 'csv-linter',
    description = 'lint csv files',
    packages = find_packages(),
    author = 'Alfredo Deza',
    entry_points="""
    [console_scripts]
    csv-linter=csv_linter:main
    """,
    install_requires = ['click==7.1.2', 'pandas==1.2.0'],
    version = '0.0.1',
    url = 'https://github.com/paiml/practical-mlops-book',
)
```

This file allows Python installers to capture all the details of the Python package like dependencies and, in this case, the availability of a new command line tool called *csv-linter*. Most of the fields in the `setup` call are straightforward, but it is worth noting the details of the `entry_points` value. This is a feature of the *setuptools* library, which allows defining a function within a Python file to map back to a command line tool name. In this case, I'm naming the command line tool *csv-linter*, and I'm mapping it to a function (`main`) that I'll create next inside a file called *csv_linter.py*. Although I've picked *csv-linter* to be the tool's name, it can

be named anything at all. Under the hood, the *setuptools* library will create the executable with whatever is declared here. There is no restriction to name it the same as the Python file.

Open a new file named *csv_linter.py* and add a single function that uses the Click framework:

```
import click


@click.command()
def main():
    return
```

NOTE

Even when examples don't explicitly mention using Python's virtual environments, it is always a good idea to create one. Having virtual environments is a robust way of isolating dependencies and potential problems with other libraries installed in a system.

These two files are nearly all you need to create a command line tool that (for now) doesn't do anything other than providing an executable available in your shell path. Next, create a virtual environment and activate it to install the newly created tool:

```
$ python3 -m venv venv
$ source venv/bin/activate
$ python setup.py develop
running develop
running egg_info
...
csv-linter 0.0.1 is already the active version in easy-install.pth
...
Using /Users/alfredo/.virtualenvs/practical-mlops/lib/python3.8/site-packages
Finished processing dependencies for csv-linter==0.0.1
```

The *setup.py* script has many different ways to get invoked, but you will primarily be using either the `install` argument or the `develop` one I used in the example. Using `develop` allows you to make changes to the script's source code and get those automatically available in the script, whereas `install` would create a separate (or standalone) script with no ties back to the source code. When developing command line tools, I recommend using `develop` to test changes as you make progress quickly. After calling the *setup.py* script, test out the newly available tool by passing the `--help` flag:

```
$ csv-linter --help
Usage: csv-linter [OPTIONS]

Options:
  --help  Show this message and exit.
```

Getting a help menu without having to write one is great, and it is a feature that a few other command line tool frameworks offer. Now that the tool is available as a script in the terminal, it is time to add useful features. To keep things simple, this script will accept a CSV file as a single argument. The Click framework has a built-in helper to accept files as arguments that ensure that the file exists and produces a helpful error otherwise. Update the *csv_linter.py* file to use the helper:

```
import click

@click.command()
@click.argument('filename', type=click.Path(exists=True))
def main():
    return
```

Although the script isn't doing anything with the file yet, the help menu has been updated to reflect the option:

```
$ csv-linter --help
Usage: csv-linter [OPTIONS] FILENAME
```

Still, not doing something useful. Check what happens if you pass a CSV file that doesn't exist:

```
$ csv-linter bogus-dataset.csv
Usage: csv-linter [OPTIONS] FILENAME
Try 'csv-linter --help' for help.

Error: Invalid value for 'FILENAME': Path 'bogus-dataset.csv' does not exist.
```

Take the tool a step further by using the `filename` argument that is getting passed into the `main()` function with Pandas to describe the dataset:

```
import click
import pandas as pd


@click.command()
@click.argument('filename', type=click.Path(exists=True))
def main(filename):
    df = pd.read_csv(filename)
    click.echo(df.describe())
```

The script uses Pandas, and another Click helper called *echo*, which allows us to print output back to the terminal easily. Use the *wine.csv* file previously saved when processing the dataset as input:

```
$ csv-linter wine.csv
         Unnamed: 0  grape        rating
count  32780.000000    0.0  32780.000000
mean   16389.500000    NaN     91.186608
std     9462.915248    NaN      2.190391
min        0.000000    NaN     85.000000
25%     8194.750000    NaN     90.000000
50%    16389.500000    NaN     91.000000
75%    24584.250000    NaN     92.000000
max    32779.000000    NaN     99.000000
```

Still, this isn't *too helpful*, even though it is now easily describing any CSV file using Pandas. The problem we need to solve here is to alert us about

three potential issues:

- Detect zero-count columns
- Warn when `Unnamed` columns exist
- Check if there are carriage returns in a field

Let's start with detecting zero-count columns. Pandas allows us to iterate over its columns, and has a `count()` method that we can leverage for this purpose:

```
In [10]: for key in df.keys():
    ...:     print(df[key].count())
    ...:
    ...:
32780
0
32777
32422
32780
32780
```

Adapt the loop into a function separate from `main()` in the *csv_linter.py* file so that it is isolated and keeps things readable:

```
def zero_count_columns(df):
    bad_columns = []
    for key in df.keys():
        if df[key].count() == 0:
            bad_columns.append(key)
    return bad_columns
```

The `zero_count_columns()` function takes as input the data frame from Pandas, captures all the columns with a zero-count, and returns them at the end. It is isolated and not coordinating the output with the `main()` function yet. Since it is returning a list of column names, loop over the contents of the result in the `main()` function:

```
@click.command()
@click.argument('filename', type=click.Path(exists=True))
```

```
def main(filename):
    df = pd.read_csv(filename)
    # check for zero count columns
    for column in zero_count_columns(df):
        click.echo(f"Warning: Column '{column}' has no items in it")
```

Run the script against the same CSV file (note that I removed the `.describe()` call):

```
$ csv-linter wine-ratings.csv
Warning: Column 'grape' has no items in it
```

At 19 lines, this script would've saved me a lot of time already if I had used it before sending the data into an ML platform. Next, create another function that loops over the columns to check for `Unnamed` ones:

```
def unnamed_columns(df):
    bad_columns = []
    for key in df.keys():
        if "Unnamed" in key:
            bad_columns.append(key)
    return len(bad_columns)
```

In this case, the function checks if the `"Unnamed"` string is present in the name but is not returning the names (since we assume they are all similar or even the same), but rather, it returns the total count. With that information, expand the `main()` function to include the count:

```
@click.command()
@click.argument('filename', type=click.Path(exists=True))
def main(filename):
    df = pd.read_csv(filename)
    # check for zero count columns
    for column in zero_count_columns(df):
        click.echo(f"Warning: Column '{column}' has no items in it")
    unnamed = unnamed_columns(df)
    if unnamed:
        click.echo(f"Warning: found {unnamed} columns that are Unnamed")
```

Run the tool once again against the same CSV file to check the results:

```
$ csv-linter wine.csv
Warning: Column 'grape' has no items in it
Warning: found 1 column that is Unnamed
```

Finally, and perhaps the trickiest one to detect, is finding carriage returns within a large text field. This operation can be expensive, depending on the size of the dataset. Although there are more performant ways to accomplish the iteration, the next example will try to use the most straightforward approach. Create another function that does the work over the Pandas data frame:

```
def carriage_returns(df):
    for index, row in df.iterrows():
        for column, field in row.iteritems():
            try:
                if "\r\n" in field:
                    return index, column, field
            except TypeError:
                continue
```

The loop prevents a `TypeError` from being raised. If the function does a string check against a different type, like an integer, then a `TypeError` would get produced. Since the operation can be costly, the function breaks out of the loop at the first sign of a carriage return. Finally, the loop returns the index, column, and the whole field for reporting by the `main()` function. Now update the script to include the reporting of carriage returns:

```
@click.command()
@click.argument('filename', type=click.Path(exists=True))
def main(filename):
    df = pd.read_csv(filename)
    for column in zero_count_columns(df):
        click.echo(f"Warning: Column '{column}' has no items in it")
    unnamed = unnamed_columns(df)
    if unnamed:
        click.echo(f"Warning: found {unnamed} columns that are Unnamed")
```

```
    carriage_field = carriage_returns(df)
    if carriage_field:
        index, column, field = carriage_field
        click.echo((
           f"Warning: found carriage returns at index {index}"
           f" of column '{column}':")
        )
        click.echo(f"         '{field[:50]}'")
```

Testing this last check is tricky because the dataset doesn't have carriage returns anymore. The repository for this chapter includes an example CSV with carriage returns in it. Download that file locally and point the *csv-linter* tool to that file:

```
$ csv-linter carriage.csv
Warning: found carriage returns at index 0 of column 'notes':
         'Aged in French, Hungarian, and American Oak barrel'
```

To prevent having an extremely long field printed in the output, the warning message is only showing the first 50 characters. This command line tool is leveraging the Click framework for the command line tool functionality and Pandas for the CSV inspection. Although it is doing only three checks and is not very performant, it would've been invaluable for me to prevent having issues using the dataset. There are multiple other ways to ensure a dataset is in acceptable shape, but this is an excellent example of how to automate (and prevent) problems that you encounter. Automation is the foundation of DevOps, and command line tools are an excellent way to start the automation path.

Modularizing a Command Line Tool

The previous command line tool showed how to use Python's internal libraries to create a script from a single Python file. But it is entirely possible to use a directory with multiple files that make up a single command line tool. This approach is preferable when the content of a single script starts getting hard to read. There is no hard limit on what should make you split a long file into multiple ones; I recommend grouping code that

shares common responsibilities and separating them, especially when there is a need for code reuse. There might not be a use case for code re-use in some situations, but splitting some pieces could still make sense to improve readability and maintenance.

Let's reuse the example of the *csv-linter* tool to adapt the single-file script into multiple files in a directory. The first step is creating a directory with an *__init__.py* file and move the *csv_linter.py* file into it. Using an *__init__.py* file tells Python to treat that directory as a module. The structure should now look like this:

```
$ tree .
.
├── csv_linter
│   ├── __init__.py
│   └── csv_linter.py
├── requirements.txt
└── setup.py

1 directory, 4 files
```

At this point, there is no need to repeat the name of the tool in the Python file, so rename it to something more modular and less tied to the name of the tool. I usually suggest using *main.py*, so rename the file:

```
$ mv csv_linter.py main.py
$ ls
__init__.py main.py
```

Try using the `csv_linter` command once again. The tool should be in a broken state because the files got moved around:

```
$ csv-linter
Traceback (most recent call last):
  File ".../site-packages/pkg_resources/__init__.py", line 2451, in resolve
    return functools.reduce(getattr, self.attrs, module)
AttributeError: module 'csv_linter' has no attribute 'main'
```

This is because the *setup.py* file is pointing to a module that doesn't exist anymore. Update that file so that it finds the `main()` function inside of the *main.py* file:

```
from setuptools import setup, find_packages

setup(
    name = 'csv-linter',
    description = 'lint csv files',
    packages = find_packages(),
    author = 'Alfredo Deza',
    entry_points="""
    [console_scripts]
    csv-linter=csv_linter.main:main
    """,
    install_requires = ['click==7.1.2', 'pandas==1.2.0'],
    version = '0.0.1',
    url = 'https://github.com/paiml/practical-mlops-book',
)
```

The change may be difficult to spot, but the entry point to `csv-linter` is now `csv_linter.main:main` . This change means that *setuptools* should look for a *csv_linter* package with a *main* module with a `main()` function in it. The syntax is a bit tricky to remember (I always have to look it up), but grasping the change details helps visualize how things are tied together. The installation process still has all the old references, so you must run *setup.py* again to make it all work:

```
$ python setup.py develop
running develop
Installing csv-linter script to /Users/alfredo/.virtualenvs/practical-mlops/bi
...
Finished processing dependencies for csv-linter==0.0.1
```

Now that the *csv-linter* tool is in working order again let's split up the *main.py* module into two files, one for the checks and the other one just for the command line tool work. Create a new file called *checks.py* and move the functions that do the checks from *main.py* into this new file:

```
# in checks.py

def carriage_returns(df):
    for index, row in df.iterrows():
        for column, field in row.iteritems():
            try:
                if "\r\n" in field:
                    return index, column, field
            except TypeError:
                continue


def unnamed_columns(df):
    bad_columns = []
    for key in df.keys():
        if "Unnamed" in key:
            bad_columns.append(key)
    return len(bad_columns)


def zero_count_columns(df):
    bad_columns = []
    for key in df.keys():
        if df[key].count() == 0:
            bad_columns.append(key)
    return bad_columns
```

And now update *main.py* to import the checking functions from the *checks.py* file. The newly updated main module should now look like this:

```
import click
import pandas as pd
from csv_linter.checks import
     carriage_returns,
     unnamed_columns,
     zero_count_columns

@click.command()
@click.argument('filename', type=click.Path(exists=True))
def main(filename):
    df = pd.read_csv(filename)
    for column in zero_count_columns(df):
```

```
            click.echo(f"Warning: Column '{column}' has no items in it")
        unnamed = unnamed_columns(df)
        if unnamed:
            click.echo(f"Warning: found {unnamed} columns that are Unnamed")
        carriage_field = carriage_returns(df)
        if carriage_field:
            index, column, field = carriage_field
            click.echo((
               f"Warning: found carriage returns at index {index}"
               f" of column '{column}':")
            )
            click.echo(f"         '{field[:50]}'")
```

Modularizing is a great way to keep things short and readable. When a tool separates concerns in this way, it is easier to maintain and reason about. There have been many times when I've had to work with legacy scripts that were *thousands* of lines long for no good reason. Now that the script is in good shape, we can go into microservices and take some of these concepts further.

Microservices

As I mentioned at the beginning of this chapter, microservices are a new type of application paradigm, entirely opposed to old-style monolithic applications. For ML operations, in particular, it is critical to isolate responsibilities as much as possible from the process of getting models into production. Isolating components can then pave the way for reusability elsewhere, not only tied to a particular process for a single model.

I tend to think about microservices and reusable components as pieces of a Jenga puzzle. A monolithic application would be an extremely tall Jenga tower with many pieces working together to make it stand, but with one major flaw: don't try to touch anything that will bring the whole thing down. On the other hand, if the pieces are put together as robustly as possible (as at the beginning of the puzzle game), then removing pieces and repurposing them to a different spot is straightforward.

It is common for software engineers to quickly create utilities that are tightly coupled to the task at hand. For example, some logic that removes some values from a string, which you can then use to persist it in a database. Once the few lines of code have proven their worth, it is useful to think about reusability for other code base components. I tend to have a utility module in my projects, where common utilities go so that other pieces of the application that need the same facilities can import and reuse them.

Like containerization, microservices allow concentrating more on the solution itself (the code) than the environment (e.g., operating system). One excellent solution for creating microservices is using serverless technologies. Serverless products from cloud providers take many different names (lambda and cloud functions, for example). Still, they all refer to the same thing: create a single file with some code and deploy instantly to the cloud—no need to worry about the underlying operating system or its dependencies. Just select a runtime from a drop-down menu, like Python 3.8, and click a button. Most cloud providers, in fact, allow you to create the function directly in the browser. This type of development and provisioning is fairly revolutionary, and it has been enabling interesting application patterns that were very complicated to achieve before.

Another critical aspect of serverless is that you can access most of the cloud provider's offerings without much effort. For ML, this is crucial: do you need to perform some computer vision operations? A serverless deployment can do this in less than a dozen lines of code. This way of leveraging ML operations in the cloud gets you speed, robustness, and reproducibility: all significant components of DevOps principles. Most companies will not need to create their own model for computer vision from scratch. The phrase "standing on the shoulders of giants" is a perfect fit to grasp the possibilities. Years ago, I worked in a digital media agency with a team of a dozen IT personnel who had decided to run their email servers in-house. Running email servers (correctly) takes a tremendous amount of knowledge and ongoing effort. Email is a *challenging problem to solve*. I can tell you that email would continuously stop working—in fact, it was almost a monthly occurrence.

Finally, let's look at how many options there are to build ML-based microservices on cloud providers. They typically range from more IaaS (infrastructure as a service) to more PaaS (platforms as a service). For example, in Figure 11-1, Kubernetes is a lower-level and complex technology that deploys microservices. In other scenarios, like AWS App Runner, covered earlier in the book, you can point your GitHub repo at the service and click a few buttons to get a fully deployed continuous delivery platform. Somewhere in the middle are cloud functions.

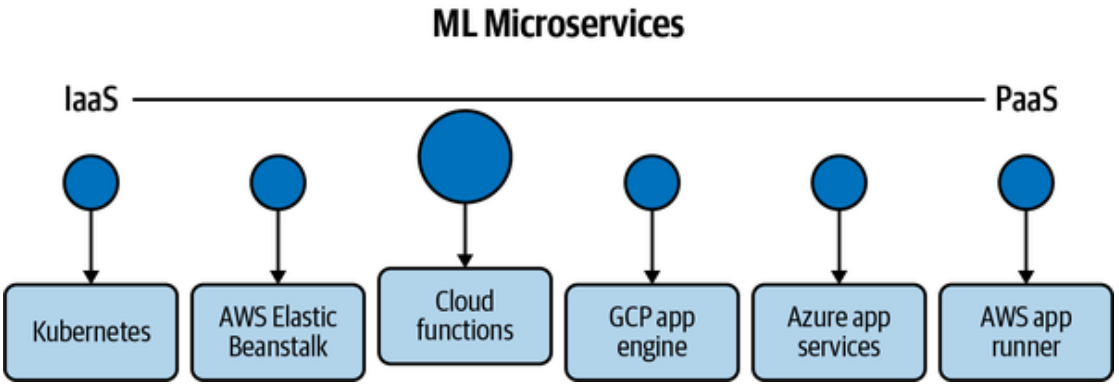

Figure 11-1. Cloud ML microservices

What is your company's core competency? If it isn't state-of-the-art computer vision models, then don't create these yourself. Likewise, work smart, not hard, and build on top of high-level systems like AWS App Runner or Google Cloud Run. Finally, resist the urge to reinvent the wheel and leverage cloud microservices.

Creating a Serverless Function

Most cloud providers expose their ML services in their serverless environments. Computer vision, Natural Language Processing, and recommendation services are just a few. In this section, you will use a translation API to leverage one of the world's most potent language-processing offerings.

NOTE

For this serverless application, I will use Google Cloud Platform (GCP). If you haven't signed up for it before, you might get some free credits to try the example in this section, although with the current limits, you should still be able to deploy the cloud function without incurring any costs.

Once logged into GCP, select Cloud Functions from the left sidebar under the Compute section as shown in Figure 11-2.

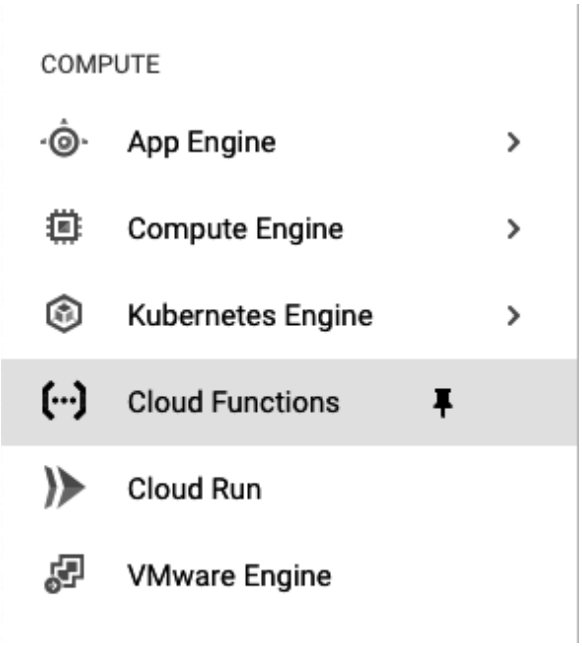

Figure 11-2. Cloud functions sidebar

If you haven't created a function before, a greeting message should show you a link to create one. If you already have deployed a function, a Create Function button should be available otherwise. Creating and deploying a function from the UI involves just a few steps. Figure 11-3 is the form you should expect to fill out.

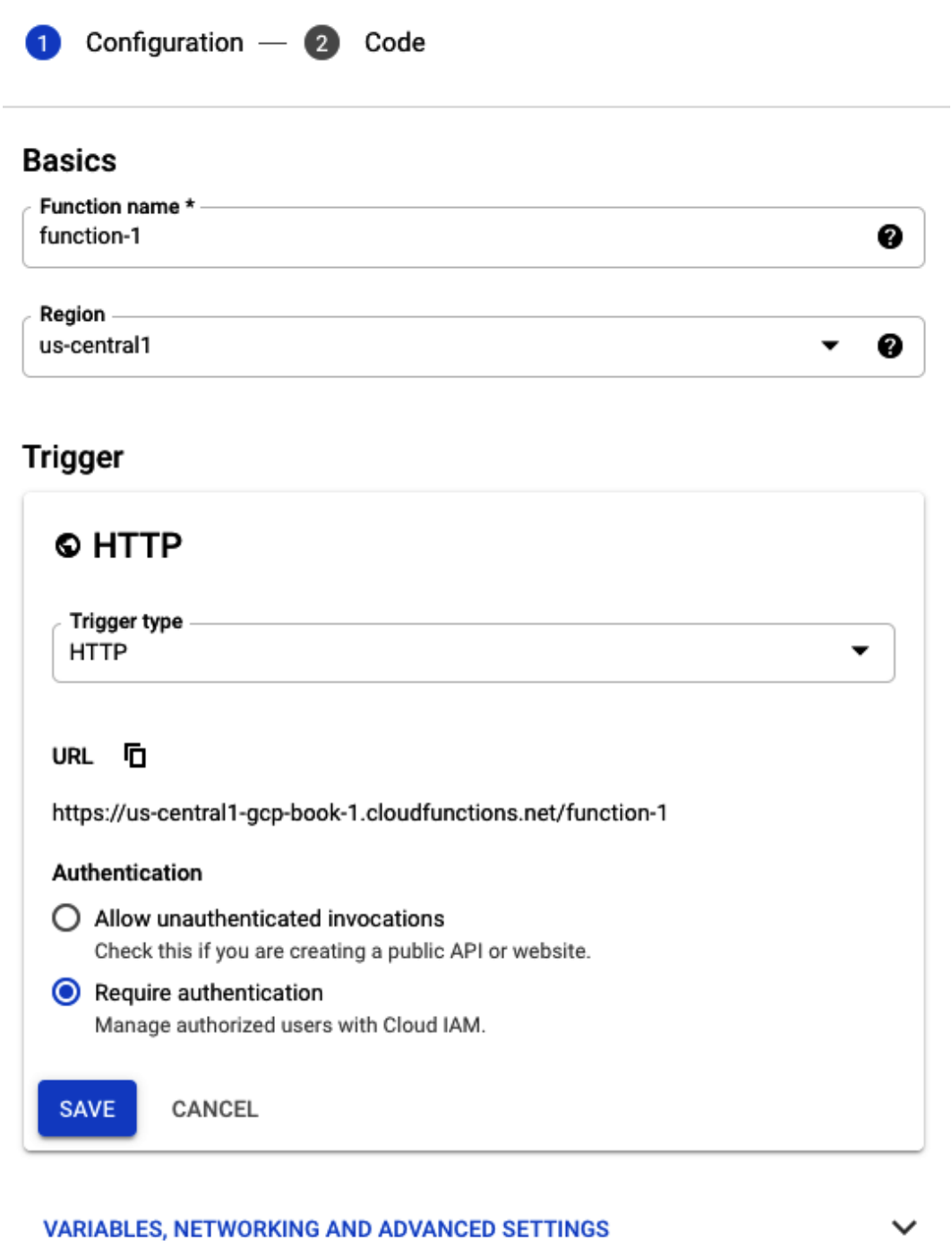

Figure 11-3. Create a cloud function

The default values for the Basics section are sufficient. In this case, the form comes pre-filled with *function-1* as the name and using *us-central1* as the region. Ensure you set the trigger type to HTTP, and that authentication is required. Click Save and then the Next button at the bottom of the page.

WARNING

Although unauthenticated invocations to functions are allowed (and as simple as selecting the option in the web form), I strongly suggest you never deploy a cloud function without authentication enabled. Exposed services over HTTP that are not authenticated pose a risk of getting abused. Since a cloud function's usage is tied directly to your account and budget, it can have a substantial financial impact from unauthorized usage.

Once in the Code section, you can select a runtime and an entry point. Select Python 3.8, change the entry point to use *main*, and update the function's name to use `main()` instead of `hello_world()` as shown in Figure 11-4.

Figure 11-4. Cloud functions code

The ability to choose the entry point for the application opens up the possibilities of creating other functions to assist the main function or determine some other naming convention to interact with the code. Flexibility is great, but having defaults and using conventions is more valuable. Once you've made the necessary changes, click the Deploy button to get this function into a production environment. Once it completes, the function should show in the Cloud Functions dashboard.

Next, after deploying, let's interact with it by sending an HTTP request. There are many ways to accomplish this. To get started, click Actions for the selected function and choose "Test function." A new page loads, and although it may be hard to see it at first, the "Triggering event" section is where you add the body of the request to be sent. Since the function is looking for a `"message"` key, update the body to include a message like Figure 11-5 shows, then click the Test the function button.

Figure 11-5. Cloud function code—Triggering event

It should only take a few seconds to get the output, which should be the output from the value of the `"message"` key. Aside from that output, some logs should show up, making this a very straightforward way of interacting with the function. One thing that was not required was doing any authentication steps, although the function was created with authentication enabled. Whenever you are debugging and want to test a deployed function quickly, this is the easiest way by far.

This function accepts JSON (JavaScript Object Notation) as input. Although it isn't clear that the cloud function uses HTTP when testing, it is how the input gets delivered to the function. JSON is sometimes referred to as the *lingua franca* (common language) of web development because programming languages and other services and implementations can consume and produce JSON to native constructs that these languages and services understand.

Although HTTP APIs can restrict what types of requests and what format the body should be, it is relatively common to use JSON to communicate. In Python, you can load JSON into native data structures like lists and dictionaries, which are straightforward to use.

Before exploring other ways to interact with the function (including authentication), let's leverage Google's ML services by using their translation service. By default, all the APIs from Google's Cloud Platform are disabled. If you need to interact with a cloud offering like language translations, you must enable the API before using it. It isn't much of an issue if you create a cloud function (as in this case) and forget to do so. The resulting behavior would be an error captured in the logs, and an HTTP 500 returned as an error response back to the client making the request. This is

an excerpt from the logs of a function that tried using the translation API without enabling it first:

```
google.api_core.exceptions.PermissionDenied: 403 Cloud Translation API has not
   been used in project 555212177956 before or it is disabled.
Enable it by visiting:
  https://console.developers.google.com/apis/api/translate.googleapis.com/
then retry. If you enabled this API recently, wait a few minutes for the
action to propagate to our systems and retry."
```

Enable the Cloud Translation API before making any further modifications to the function. Most of the APIs provided by GCP need to be enabled in a similar way, by going to the APIs and Services link and finding the API you need in the Library page.

NOTE

If you are not an admin on the GCP account and do not see an API available, you may lack the necessary permissions to enable an API. An account administrator needs to grant you proper permissions.

After enabling the API, head back to the function by clicking its name so its dashboard loads. Once in the dashboard, find the Edit button at the top of the page to make changes to the source code. The Edit section presents you first with options to configure the function itself first and then the code. There is no need to make changes to the deployment configuration, so click Next to finally get to the source. Click the *requirements.txt* link that opens that file to add the API library that is required to interact with the translation service:

```
google-cloud-translate==3.0.2
```

Now click *main.py* to edit the contents. Add the import statement to bring the translate service in, and add a new function that will be in charge of doing the translation:

```
from google.cloud import translate

def translator(text="YOUR_TEXT_TO_TRANSLATE",
    project_id="YOUR_PROJECT_ID", language="fr"):

    client = translate.TranslationServiceClient()
    parent = f"projects/{project_id}/locations/global"

    response = client.translate_text(
        request={
            "parent": parent,
            "contents": [text],
            "mime_type": "text/plain",
            "source_language_code":"en-US",
            "target_language_code":language,
        }
    )
    # Display the translation for each input text provided
    for translation in response.translations:
        print(u"Translated text: {}".format(translation.translated_text))
    return u"Translated text: {}".format(translation.translated_text)
```

This new function takes three arguments to interact with the translation API: the input text, the project ID, and the target language for the translation (defaulting to French). The input text defaults to English, but the function can be adapted to use other languages (e.g., Spanish) as input and English as the output. As long as the language is supported, the function can use the input and output with any combination.

The translation request response is iterable, so a loop is needed after the translation is complete.

Now modify the `main()` function to pass the value from `"message"` into the `translator()` function. I'm using my own project ID (*"gcp-book-1"*) so make sure you update that with your own if trying the next example:

```
def main(request):
    request_json = request.get_json()
    if request_json and 'message' in request_json:
        return translator(
```

```
            text=request_json['message'],
            project_id="gcp-book-1"
        )
    else:
        return f'No message was provided to translate'
```

The `main()` function still requires a `"message"` value assigned in the incoming JSON request but will now do something useful with it. Test it out in the console with a sample JSON input:

```
{"message": "a message that has been translated!"}
```

The output in the test page (shown in Figure 11-6) should be almost immediate.

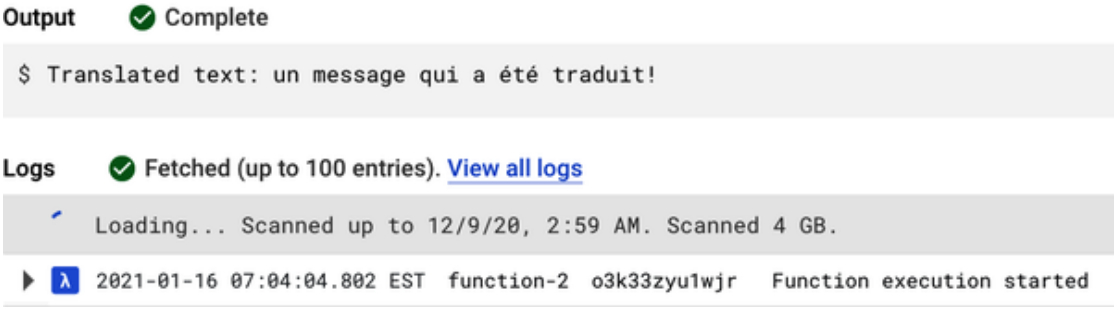

Figure 11-6. Translated test

Authenticating to Cloud Functions

I see HTTP access as access democracy: tremendous flexibility for other systems and languages to interact from their implementations into a separate service in a remote location using the HTTP spec. All major programming languages can construct HTTP requests and process responses from servers. Bringing services together with the help of HTTP allows these services to work in new ways, potentially in ways that were not thought of initially. Think of HTTP APIs as extensible functionality that can plug into anything connected to the internet. But connecting over the internet has security implications, like preventing unauthorized access with authenticated requests.

There are a few ways you can interact remotely with a cloud function. I'll start with the command line, using the *curl* program. Although I tend not

to use *curl* for interacting with authenticated requests, it does offer a straightforward way of documenting all the pieces that you need to be successful in submitting the request. Install the Google Cloud SDK for your system, and then determine the project ID and the function name you deployed before. The following example uses *curl* with the SDK to authenticate:

```
$ curl -X POST --data '{"message": "from the terminal!"}' \
  -H "Content-Type: application/json" \
  -H "Authorization: bearer $(gcloud auth print-identity-token)" \
  https://us-central1-gcp-book-1.cloudfunctions.net/function-1
```

On my system, using the *function-1* URL, I get the following response:

```
Translated text: du terminal!
```

The command seems very involved, but it presents a better picture of what is needed to make a successful request. First, it declared that the request uses a POST method. This method is commonly used when a payload is associated with a request. In this case, `curl` is sending JSON from the argument to the `--data` flag. Next, the command adds two request headers, one to indicate the type of content being sent (JSON) and the other one to indicate that the request is providing a token. The token is where the SDK comes into play because it creates a token for the request, which is required by the cloud function service to verify the request as authenticated. Finally, the URL for the cloud function is used as the target for this authenticated POST request sending JSON.

Try running the SDK command on its own to see what it does:

```
$ gcloud auth print-identity-token
aIWQo6IClq5fNylHWPHJRtoMu4IG0QmP84tnzY5Ats_4XQvClne-A9coqEciMu_WI4Tjnias3fJjal
[...]
```

Now that you understand the required components for the request, try the SDK directly to make the request for you to reach the deployed cloud function:

```
$ gcloud --project=gcp-book-1 functions call function-1 \
  --data '{"message":"I like coffee shops in Paris"}'
executionId: 1jgd75feo29o
result: "Translated text: J'aime les cafés à Paris"
```

I think it is an excellent idea from cloud providers like Google to include other facilities to interact with their services as it happens here with the cloud function. If you only were aware of the SDK command to interact with a cloud function, it would be difficult to use a programming language to construct a request, for example, with Python. These options offer flexibility, and the more flexible an environment is, the better the chances of adapting it in the most reasonable way to fit your environment's needs.

Now let's use Python to interact with the translator function.

NOTE

The following example will call out directly to the `gcloud` command using Python, making it easier to quickly demonstrate how to create Python code to interact with the cloud function. However, it isn't a robust way of dealing with authentication. You will need to create a service account and use the *google-api-python-client* to secure the authentication process properly.

Create a Python file called *trigger.py* and add the following code to retrieve the token from the `gcloud` command:

```
import subprocess


def token():
    proc = subprocess.Popen(
        ["gcloud",  "auth", "print-identity-token"],
        stdout=subprocess.PIPE)
    out, err = proc.communicate()
    return out.decode('utf-8').strip('\n')
```

The `token()` function will call the `gcloud` command and process the output to make the request. It is worth reiterating that this is a quick way to demonstrate making requests to trigger the function from Python. You should consider creating a Service Account and OAuth2 from the *google-api-python-client* if looking to implement this in a production environment.

Now create the request using that token to communicate with the cloud function:

```
import subprocess
import requests

url = 'https://us-central1-gcp-book-1.cloudfunctions.net/function-1'


def token():
    proc = subprocess.Popen(
        ["gcloud",  "auth", "print-identity-token"],
        stdout=subprocess.PIPE)
    out, err = proc.communicate()
    return out.decode('utf-8').strip('\n')


resp = requests.post(
    url,
    json={"message": "hello from a programming language"},
    headers={"Authorization": f"Bearer {token()}"}
)

print(resp.text)
```

Note that I've added the *requests* library (version 2.25.1 in my case) to the script, so you need to install it before continuing. Now run the *trigger.py* file to test it out, ensuring you have updated the script with your project ID:

```
$ python trigger.py
Translated text: bonjour d'un langage de programmation
```

Building a Cloud-Based CLI

Now that you understand the concepts for building a command line tool, packaging it, and distributing it while leveraging the cloud for its ML offerings, it is interesting to see these come together. In this section, I will reuse all the different parts to create one. Create a new directory, and add the following to a *setup.py* file so that packaging is solved right away:

```
from setuptools import setup, find_packages

setup(
    name = 'cloud-translate',
    description = "translate text with Google's cloud",
    packages = find_packages(),
    author = 'Alfredo Deza',
    entry_points="""
    [console_scripts]
    cloud-translate=trigger:main
    """,
    install_requires = ['click==7.1.2', 'requests==2.25.1'],
    version = '0.0.1',
    url = 'https://github.com/paiml/practical-mlops-book',
)
```

The *setup.py* file will create a *cloud-translate* executable mapped to a `main()` function within the *trigger.py* file. We haven't created that function yet, so add the *trigger.py* file that was created in the previous section and add the function:

```
import subprocess
import requests
import click

url = 'https://us-central1-gcp-book-1.cloudfunctions.net/function-2'


def token():
    proc = subprocess.Popen(
        ["gcloud",  "auth", "print-identity-token"],
        stdout=subprocess.PIPE)
```

```
    out, err = proc.communicate()
    return out.decode('utf-8').strip('\n')


@click.command()
@click.argument('text', type=click.STRING)
def main(text):
    resp = requests.post(
            url,
            json={"message": text},
            headers={"Authorization": f"Bearer {token()}"})

    click.echo(f"{resp.text}")
```

The file isn't that different from the initial *trigger.py* where it runs directly with Python. The Click framework allows us to define a *text* input and then print the output to the terminal when it completes. Run `python setup.py develop` so that everything gets wired together, including the dependencies. As expected, the framework gives us the help menu:

```
$ cloud-translate --help
Usage: cloud-translate [OPTIONS] TEXT

Options:
  --help  Show this message and exit.
$ cloud-translate "today is a wonderful day"
Translated text: aujourd'hui est un jour merveilleux
```

Machine Learning CLI Workflows

The shortest distance between two points is a straight line. Likewise, a command line tool is often the most straightforward approach to using machine learning. In Figure 11-7, you can see that there are many different styles of ML techniques. In the case of unsupervised machine learning, you can train "on the fly"; in other cases, you may want to use a trained nightly model and place it in object storage. Yet, you may wish to use high-level tools like AutoML, AI APIS, or models created by third parties in others.

Notice that there are many different problem domains where adding ML enhances a CLI or is the entire purpose of the CLI. These domains include text, computer vision, behavioral analytics, and customer analysis.

It is essential to point out that there are many targets for deploying command line tools that include machine learning. A filesystem like Amazon EFS, GCP Filestore, or Red Hat Ceph have the advantage of being a centralized Unix mount point for a cluster. The *bin* directory could include ML CLI tools delivered via a Jenkins server that mounts this same volume.

Other delivery targets include the Python Package Repository (PyPI), and public Container Registries like Docker, GitHub, and Amazon. Yet more targets include Linux packages like Debian and RPM. A command line tool that packages machine learning has a much more extensive collection of deployment targets than even a microservice because a command line tool is an entire application.

Figure 11-7. Machine learning CLI workflows

A few good examples of projects suitable for doing machine learning with a CLI include the following resources:

DevML

DevML is a project that analyzes GitHub organizations and allows ML practitioners to create their own "secret ML" predictions, perhaps by connecting it to streamlit or including developer clustering reports in Amazon QuickSight.

Python MLOps Cookbook

The Python MLOps Cookbook GitHub repo contains a simple ML model as a set of utilities. This project has detailed coverage in Chapter 7.

Spot Price Machine Learning

> Yet another ML CLI example is a project on Spot Price Machine Learning Clustering. In this GitHub repository, different attributes of AWS spot instances, including memory, CPU, and price, are used to create clusters of machine types that are similar.

With these CLI workflows out of the way, let's move on to wrapping up the chapter.

Conclusion

This chapter has described how to create command line tools from the ground up and use a framework to create tools to perform automation rapidly. Even when examples may look trivial, the details and how things work together are essential aspects. When learning new concepts or subjects that others commonly dread (like packaging in general), it is easy to feel discouraged and try to work around them. Although Python has a long road toward better packaging, getting started is not that hard, and doing the heavy lifting with proper packaging of your tools will make you invaluable in any team. With packaging and command line tools, you are now well-positioned to start bringing different services together for automation.

This chapter did that by leveraging the cloud and its many ML offerings: a powerful translation API from Google. It is critical to remember that there is no need to create all the models from scratch and you should leverage cloud providers' offerings whenever possible, especially when it isn't a core competency of your company.

Finally, I want to emphasize that being able to craft new solutions to tricky problems is an MLOps superpower, based on knowing how to connect services and applications. As you now know, using HTTP, command line tools, and leveraging cloud offerings via their SDKs is a strong foundation to make substantial improvements in almost any production environment.

In the next chapter we go into other details of machine learning engineering, and one of my favorite topics: case studies. Case studies are real-

world problems and situations where you can extract useful experiences and apply them today.

Exercises

- Add some more options to the CLI that use cloud functions, like making the URL configurable.
- Find out how the Service Account and OAuth2 work with the Google SDK and integrate it in *trigger.py* to avoid using the *subprocess* module.
- Enhance the cloud function by translating a separate source, like a page from Wikipedia.
- Create a new cloud function that does image recognition and make it work with a command line tool.
- Fork the Python MLOps Cookbook repository and build a slightly different containerized CLI tool that you publish to a public container registry like DockerHub or GitHub Container Registry.

Critical Thinking Discussion Questions

- Name one possible consequence of unauthenticated cloud functions.
- What are some of the drawbacks of not using a virtual environment?
- Describe two aspects of good debugging techniques and why they are useful.
- Why is knowing packaging useful? What are some critical aspects of packaging?
- Is it a good idea to use an existing model from a cloud provider? Why?
- Explain the trade-offs in deploying an open source CLI tool powered by machine learning using a public container registry versus using the Python Package Repository.