From Jupyter notebook prototypes of Machine Learning systems to the production system on the cloud

Notebooks, like Jupyter Notebooks, are interactive computing environments that allow you to write and execute code in a step-by-step, cell-based format—alongside rich text, visualizations, and output—all in a single document.

They support live code, equations, visualizations, and narrative text, making them especially popular in data science, machine learning, and research. They are most commonly used with Python.

Notebooks are useful for system prototyping in machine learning systems because they allow for quick testing of ideas, visualizing results, and iterating over code for refinement.

It's tempting to keep using notebooks even when moving toward production, especially if everything seems to "just work" during development. But for production-grade systems, especially when running in the cloud using containers, there are strong reasons to refactor your notebook code into a proper Python project.

Notebooks are great for prototyping, but they lack the structure, robustness, and scalability required for production. In contrast, a well-structured Python project integrates better with CI/CD, testing, logging, container orchestration, monitoring, and other production needs.

Reasons You Need a Python Project for Production (Especially Cloud + Containers):

1. Modularity and Maintainability. Python projects organize code into modules and packages, making it easy to reuse, test, and maintain, while notebooks often contain messy, linear code that's hard to reuse across different components (e.g., inference, training, preprocessing).
2. Testing and Validation. Python projects allow unit and integration testing using tools like pytest.
3. CI/CD Compatibility. Code in Python projects can be version-controlled, linted, tested, and deployed automatically with CI/CD pipelines (e.g., GitHub Actions, GitLab CI). Containerized apps (Docker) often rely on clean, scriptable entry points (train.py, predict.py, serve.py)—not notebooks.
4. Packaging and Dependency Management. A Python project defines environments explicitly with requirements.txt, setup.py, etc. Notebooks typically rely on the user's current environment, which can lead to "it works on my machine" problems.
5. Cloud and Container Integration. In cloud environments, code often runs in containers.
6. Team Collaboration. Collaborating on notebooks is difficult
7. Serving Models (APIs). You can't serve a notebook. You need a project structure to Expose models via REST APIs (e.g., using FastAPI/Flask)

This is a typical structure of a notebook is a JSON with cells of type “markdown” for the formatted text and explanations, and “code” for the Python code:

```
{
  "cells": [
    {
      "cell_type": "markdown",
      "metadata": {},
      "source": [
        "## This is a markdown cell\n",
        "It contains *formatted* text and explanations."
      ]
    },
    {
      "cell_type": "code",
      "execution_count": 1,
      "metadata": {},
      "outputs": [
        {
          "output_type": "stream",
          "name": "stdout",
          "text": ["Hello, world!\n"]
        }
      ],
      "source": [
        "print('Hello, world!')"
      ]
    },
    {
      "cell_type": "code",
      "execution_count": 2,
      "metadata": {},
      "outputs": [
        {
          "output_type": "execute_result",
          "data": {
            "text/plain": ["4"]
          },
          "execution_count": 2
        }
      ],
      "source": ["2 + 2"]
```

The typical folder structure in Machine Learning projects in Python

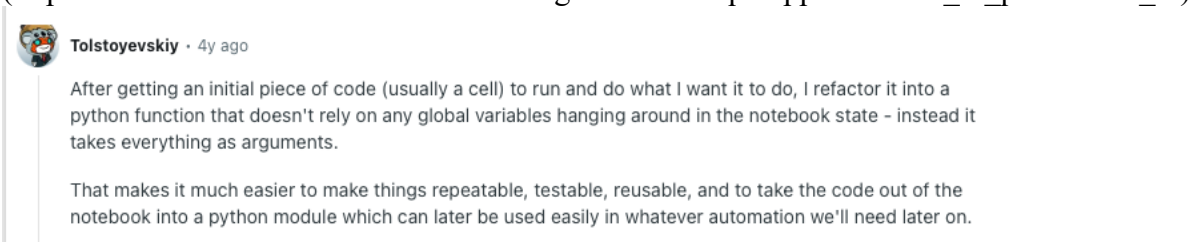
```
project_name/
│
├── data/                    # Raw and processed data
│   ├── raw/                 # Unmodified data (input)
│   ├── processed/           # Cleaned/feature-engineered data
│   └── external/            # Third-party data (e.g., external datasets)
│
├── src/                     # All source code for the project
│   ├── __init__.py
│   ├── data/                # Data loading and preprocessing
│   │   ├── load_data.py
│   │   └── preprocess.py
│   ├── features/            # Feature engineering scripts
│   │   └── build_features.py
│   ├── models/              # Model training and evaluation
│   │   ├── train_model.py
│   │   └── evaluate_model.py
│   ├── predict/             # Scripts for inference/prediction
│   │   └── predict.py
│   └── utils/               # Utility functions
│       └── helpers.py
│
├── models/                  # Serialized models (e.g., .pkl, .joblib)
│   └── model_v1.pkl
│
├── outputs/                 # Evaluation results, predictions, or plots
│   ├── figures/
│   └── predictions/
│
├── config/                  # Configuration files (YAML/JSON)
│   └── config.yaml
│
├── tests/                   # Unit and integration tests
│   └── test_train_model.py
│
├── requirements.txt         # Python dependencies
├── environment.yml          # Conda environment (optional)
├── .gitignore
├── README.md
└── setup.py                 # For packaging (optional)
```

Or, in the MLops area, deploying it into the cloud:

```
project_name/
│
├── data/                          # Data sources and versioning
│   ├── raw/
│   ├── processed/
│   └── external/
│
├── src/                           # Source code
│   ├── __init__.py
│   ├── data/                      # Loading and preprocessing
│   ├── features/                  # Feature engineering
│   ├── models/                    # Training and evaluation
│   ├── predict/                   # Inference pipeline
│   ├── serving/                   # Model API or batch deployment logic
│   ├── utils/                     # Reusable helpers
│   └── monitoring/               # Drift detection, metrics, logging
│
├── configs/                       # Configuration files
│   ├── config.yaml                # Paths, hyperparameters, etc.
│   ├── logging.yaml               # Logging setup
│   └── model_config.yaml          # Model architecture, training config
│
├── models/                        # Saved models
│   ├── latest/
│
├── outputs/                       # Metrics, plots, reports, predictions
│
├── dvc.yaml                       # DVC pipeline definition
├── .dvc/                          # DVC cache and config
├── .env                           # Environment variables (never push to git)
│
├── scripts/                       # CLI scripts (e.g., train.py,predict.py)
│   ├── train.py
│   ├── predict.py
│   └── serve_model.py
│
├── ci/                            # CI/CD pipelines (GitHub Actions, GitLab CI,
etc.)
│   └── github_actions.yml
│
├── deployment/                    # Deployment configs
│   ├── docker/                    # Docker-related files
│   │   └── Dockerfile
│   └── fastapi/                   # FastAPI/Flask for REST API serving
│
├── tests/                         # Unit and integration tests
│   └── test_model.py
│
├── logs/                          # Logs for training/inference
│
├── requirements.txt               # Python dependencies
├── environment.yml                # Conda environment
├── setup.py                       # For packaging the project
├── README.md
```

How can we obtain a project structure with the code starting from a notebook?

One can get inspired by Tolstoyevskiy's manual approach in 2021: (https://www.reddit.com/r/MachineLearning/comments/q344pp/notebook_to_production_d/):



Tolstoyevskiy · 4y ago

After getting an initial piece of code (usually a cell) to run and do what I want it to do, I refactor it into a python function that doesn't rely on any global variables hanging around in the notebook state - instead it takes everything as arguments.

That makes it much easier to make things repeatable, testable, reusable, and to take the code out of the notebook into a python module which can later be used easily in whatever automation we'll need later on.

Can you automate it by getting the help of Metamodels, models, model-to-model transformations, and model-to-text transformations?