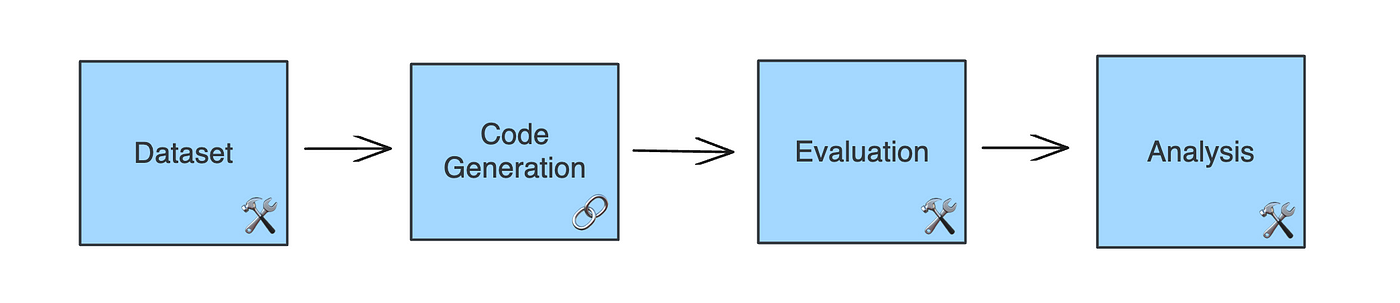
**Evaluating code generation agents—LangChain and CodeChain**

<https://medium.com/@jamesmurdza/evaluating-llms-on-code-generation-langchain-and-codechain-5a804cb1e31c>

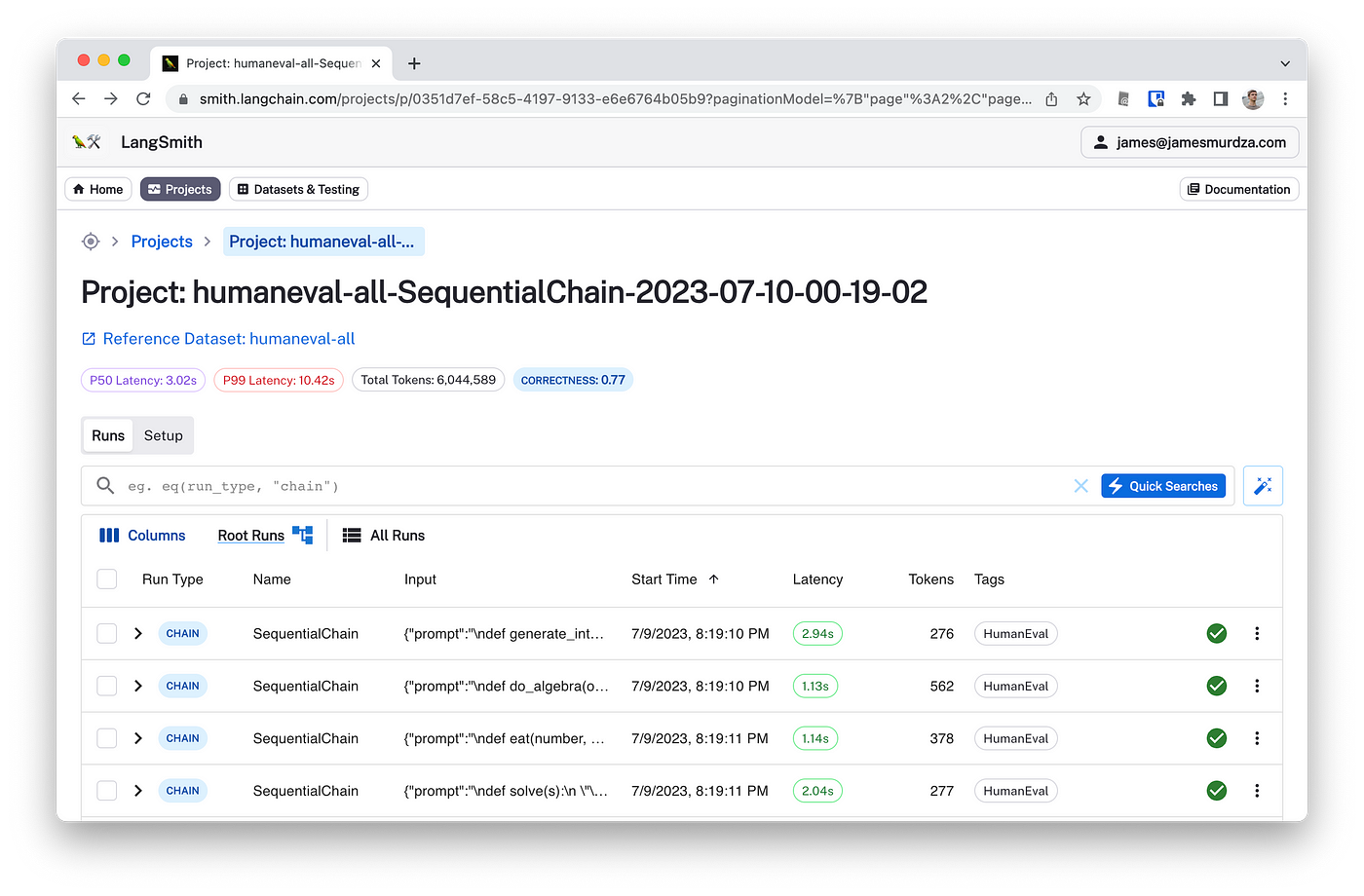
A few months ago while working on [GitWit](https://gitwit.dev/" \t "_blank), an AI code generation agent, I ran into the challenge: *How can the quality of code generated by LLM agents be accurately assessed?* Such agents are complex, non-deterministic, and often require special evaluation solutions such as running unit tests. Below is a walkthrough of my own system for dealing with this, including the code and links to the tools you need to run it yourself.



A code generation evaluation pipeline.

In the walkthrough I’ll first show how LLMs can easily be used to generate code. Then, I’ll show how I’m using [LangSmith](https://smith.langchain.com/" \t "_blank) as a platform to batch evaluate thousands of generations, which is essential if improvements in these models are to be accurately measured.

If you can’t wait to get started and want to run GPT-4 on an entire dataset of coding problems, you can get started right away with [this Colab notebook](https://github.com/jamesmurdza/langchain-humaneval)!



An automated test run of HumanEval on [LangSmith](https://smith.langchain.com/" \t "_blank) with 16,000 code generations.

**Technologies used**

Before getting started, some of the most important components in the evaluation workflow:

* [**LangChain**](https://www.langchain.com/) 🦜🔗 is a standard framework for building LLM apps and agents. Once a program is written with LangChain, its normally quite easy to swap out various components and LLM.
* [**LangSmith**](https://smith.langchain.com/) 🦜🛠️ is a data platform for applications built with LangChain. It comes with a client to batch run LLM chains and evaluators, which you define in Python code.
* Additionally, I’m using **[HumanEval](https://github.com/jamesmurdza/human-eval" \t "_blank)**, a test harness by OpenAI, and **[CodeChain](https://github.com/jamesmurdza/codechain" \t "_blank)**, a library I created myself to abstract common tasks in code generation.

**A simple code generator**

Let’s start with a simple case similar to the ones we want to evaluate. By composing a CompleteCodeChain with an ChatOpenAI, we have a working example of a code generator:

from codechain.generation import CompleteCodeChain  
from langchain.chat\_models import ChatOpenAI  
  
generator = CompleteCodeChain.from\_llm(  
 ChatOpenAI(model="gpt-3.5-turbo", temperature=0.2)  
 )  
  
result = await generator.arun("""  
def is\_palindrome(string: str):  
 # Check if string is a palindrone.  
""")

In the example, we prompt the generator with the beginning of a function that should calculate if a string is a palindrome. The result is great:

def is\_palindrome(string: str):  
 # Check if string is a palindrome.  
 return string == string[::-1]

Because of the composable nature of LangChain, it’s easy to change out the OpenAI model with one of many possible [chat integrations](https://python.langchain.com/docs/integrations/chat/).

**Creating a problem dataset**

The HumanEval dataset is a collection of Python problems, each in the same format as the example above. Each one has an ID, a prompt, and unit tests to automatically verify any attempts at a solution. They also have a canonical solution, which is purely for reference.

Creating a dataset with LangSmith is fairly straightforward. In the code below, we copy the fields we need into a dataset tied to our account on the cloud platform:

import langsmith  
from human\_eval.data import read\_problems  
  
# Connect to LangSmith and create a dataset.  
client = langsmith.Client()  
dataset = client.create\_dataset(dataset\_name, description=description)  
  
# Upload each item from HumanEval to the LangSmith dataset.  
for key, value in read\_problems().items():  
 client.create\_example(  
 inputs={ "prompt": value["prompt"], "task\_id": key },  
 outputs={ "canonical\_solution": value["canonical\_solution"] },  
 dataset\_id=dataset.id  
 )

**Defining a evaluator**

As mentioned, each item in the problem set includes a set of test cases to check if the generated code is correct. In OpenAI’s HumanEval evaluator, these assert() statements are appended onto the end of the generated code, and then the code is run, checking for success. To use this custom evaluation function with LangSmith, we subclass the RunEvaluator:

from langsmith.evaluation import RunEvaluator, EvaluationResult  
from langsmith.schemas import Run, Example  
  
from human\_eval.execution import check\_correctness  
  
class HumanEvalEvaluator(RunEvaluator):  
 def evaluate\_run(self, run: Run, example: Optional[Example] = None) -> EvaluationResult:  
 problem = problems[run.inputs["task\_id"]]  
 solution = run.outputs["output"]  
 result = check\_correctness(problem, solution, timeout=5)  
 return EvaluationResult(  
 key = "Correctness",  
 score = bool(result["passed"])  
 )

**Running a batch evaluation**

Similarly to the simple example above, we will define a code generator to run the dataset on. The HumanEvalChain is similar to CompleteCodeChain, but adapted for the input fields defined by the HumanEval dataset.

from langchain.chat\_models import ChatOpenAI  
from codechain.generation import HumanEvalChain  
  
# Factory for the generation chain  
def chain\_factory():  
 return HumanEvalChain.from\_llm(  
 ChatOpenAI(model\_name=model\_name, temperature=temperature)  
 )

Now that a dataset has been created, and the generator and evaluator have been defined, all that is left to do is run the batch evaluation. LangChain’s **arun\_on\_dataset()** runs a chain across a dataset in parallel adding evaluation results in real-time as each instance finishes.

from langchain.smith import arun\_on\_dataset, RunEvalConfig  
  
# Evaluator configuration  
evaluation = RunEvalConfig(  
 custom\_evaluators=[HumanEvalEvaluator()],  
 input\_key="task\_id"  
 )  
  
# Run all generations and evaluations  
chain\_results = await arun\_on\_dataset(  
 client=client,  
 dataset\_name=dataset\_name,  
 num\_repetitions=repetitions\_per\_problem,  
 concurrency\_level=5,  
 llm\_or\_chain\_factory=chain\_factory,  
 evaluation=evaluation,  
 tags=["HumanEval"],  
 verbose=True  
)

Running this with ten repetitions per problem will produce 1,640 generations and evaluations. To try it, run [this notebook](https://colab.research.google.com/github/jamesmurdza/humaneval-langchain/blob/main/HumanEval_with_LangChain.ipynb).

**Source code**

* [humaneval-langchain](https://github.com/jamesmurdza/humaneval-langchain/): A working example of the code shown in this post.
* [humaneval-results](https://github.com/jamesmurdza/humaneval-results/): Results and analysis from running this workflow.