📊 Day 5: Evaluation

Welcome to day five of our AI Agents Crash Course.

Yesterday we learned about function calling and created our first agent using Pydantic AI.

But is this agent actually good? Today we will see how to answer this question.

In particular, we will cover:

- Build a logging system to track agent interactions
- Create automated evaluation using AI as a judge
- Generate test data automatically
- Measure agent performance with metrics

At the end of this lesson, you'll have a thoroughly tested agent with performance metrics.

In this lesson, we'll use the FAQ database with text search, but it's applicable for any other use case.

This is going to be a long lesson, but an important one. Evaluation is critical for building reliable AI systems. Without proper evaluation, you can't tell if your changes improve or hurt performance. You can't compare different approaches. And you can't build confidence before deploying to users.

So let's start!

Logging

The easiest thing we can do to evaluate an agent is interact with it. We ask something and look at the response. Does it make sense? For most cases, it should.

This approach is called "vibe check" - we interact with it, and if we like the results, we go ahead and deploy it.

If we don't like something, we go back and change things:

- Maybe our chunking method is not suitable? Maybe we need to have a bigger window size?

- Is our system prompt good? Maybe we need more precise instructions?
- Or we want to change something else

And we iterate.

It might be okay for the first MVP, but how can we make sure the result at the end is actually good?

We need systematic evaluation. Manual testing doesn't scale - you can't manually test every possible input and scenario. With systematic evaluation, we can test hundreds or thousands of cases automatically.

We also need to base our decisions on data. It will help us to

- Compare different approaches
- Track improvements
- Identify edge cases

We can start collecting this data ourselves: start with vibe checking, but be smart about it. We don't just test it, but also record the results.

Here's the agent we created yesterday:

```python
from typing import List, Any
from pydantic_ai import Agent


def text_search(query: str) -> List[Any]:
    """
    Perform a text-based search on the FAQ index.

    Args:
        query (str): The search query string.

    Returns:
        List[Any]: A list of up to 5 search results returned by the FAQ index.
    """
    return faq_index.search(query, num_results=5)


system_prompt = """
You are a helpful assistant for a  course.
```

```
Use the search tool to find relevant information from the course materials
before answering questions.

If you can find specific information through search, use it to provide accurate
answers.
If the search doesn't return relevant results, let the user know and provide
general guidance.
"""

from pydantic_ai import Agent

agent = Agent(
    name="faq_agent",
    instructions=system_prompt,
    tools=[text_search],
    model='gpt-4o-mini'
)
```

Let's interact with the agent:

```python
question = "how do I install Kafka in Python?"
result = await agent.run(user_prompt=question)
```

Here's what we want to record:

- The system prompt that we used
- The model
- The user query
- The tools we use
- The responses and the back-and-forth interactions between the LLM and our tools
- The final response

To make it simpler, we'll implement a simple logging system ourselves: we will just write logs to json files.

You shouldn't use it in production. In practice, you will want to send these logs to some log collection system, or use specialized LLM evaluation tools like Evidently, LangWatch or Arize Phoenix.

Let's extract all this information from the agent and from the `run` results:

```python
from pydantic_ai.messages import ModelMessagesTypeAdapter


def log_entry(agent, messages, source="user"):
    tools = []

    for ts in agent.toolsets:
        tools.extend(ts.tools.keys())

    dict_messages = ModelMessagesTypeAdapter.dump_python(messages)

    return {
        "agent_name": agent.name,
        "system_prompt": agent._instructions,
        "provider": agent.model.system,
        "model": agent.model.model_name,
        "tools": tools,
        "messages": dict_messages,
        "source": source
    }
```

This code extracts the key information from our agent:

- the configuration (name, prompt, model)
- available tools
- complete message history (user input, tool calls, responses)

We also use `ModelMessagesTypeAdapter.dump_python(messages)` to convert internal message format into regular Python dictionaries. This makes it easier to save it to JSON and process later.

We also add the `source` parameter. It tracks where the question came from. We start with "user" but later we'll use AI-generated queries. Sometimes it may be important to tell them apart for analysis.

This code is generic so it will work with any Pydantic AI agent. If you use a different library, you'll need to adjust this code.

Let's write these logs to a folder:

```python
import json
import secrets
from pathlib import Path
from datetime import datetime


LOG_DIR = Path('logs')
LOG_DIR.mkdir(exist_ok=True)


def serializer(obj):
    if isinstance(obj, datetime):
        return obj.isoformat()
    raise TypeError(f"Type {type(obj)} not serializable")


def log_interaction_to_file(agent, messages, source='user'):
    entry = log_entry(agent, messages, source)

    ts = entry['messages'][-1]['timestamp']
    ts_str = ts.strftime("%Y%m%d_%H%M%S")
    rand_hex = secrets.token_hex(3)

    filename = f"{agent.name}_{ts_str}_{rand_hex}.json"
    filepath = LOG_DIR / filename

    with filepath.open("w", encoding="utf-8") as f_out:
        json.dump(entry, f_out, indent=2, default=serializer)

    return filepath
```

This code:

- Creates a logs directory (if not created previously)
- Generates unique filenames with timestamp and random hex
- Saves complete interaction logs as JSON files
- Handles datetime serialization (using the `serialized` function)

Now we can interact with it and do some vibe checking:

```python
question = input()
result = await agent.run(user_prompt=question)
print(result.output)
log_interaction_to_file(agent, result.new_messages())
```

This creates a simple interactive loop where:

- User enters a question
- Agent processes it and responds
- Complete interaction is logged to a file

Try these questions:

- how do I use docker on windows?
- can I join late and get a certificate?
- what do I need to do for the certificate?

Adding References

When interacting with the agent, I noticed one thing: it doesn't include the reference to the original documents.

Let's fix it by adjusting the prompt:

```python
system_prompt = """
You are a helpful assistant for a course.

Use the search tool to find relevant information from the course materials
before answering questions.

If you can find specific information through search, use it to provide accurate
answers.

Always include references by citing the filename of the source material you
used.
When citing the reference, replace "faq-main" by the full path to the GitHub
repository: "https://github.com/DataTalksClub/faq/blob/main/"
Format: [LINK TITLE](FULL_GITHUB_LINK)
```

```
If the search doesn't return relevant results, let the user know and provide
general guidance.
""".strip()

# Create another version of agent, let's call it faq_agent_v2
agent = Agent(
    name="faq_agent_v2",
    instructions=system_prompt,
    tools=[text_search],
    model='gpt-4o-mini'
)
```

This is the output I now get for the question "can I join late and get a certificate?":

> Yes, you can join the course late and still be eligible for a certificate, as long as you complete the required peer-reviewed capstone projects on time. You do not need to complete the homework assignments if you join late, which allows for flexibility in participation.
>
> However, please note that certificates are only awarded to those who finish the course with a “live” cohort; they are not available for those who choose the self-paced mode. This is because peer-reviewing capstone projects is a requirement that can only be done while the course is active.
>
> For more details, you can refer to the following resources:
>
> - Do I need to do the homeworks to get the certificate?
> - Can I follow the course in a self-paced mode and get a certificate?
> - Can I still join the course after the start date?

Note that I added this to the prompt:

> When citing the reference, replace "faq-main" by the full path to the GitHub repository: "https://github.com/DataTalksClub/faq/blob/main/"

When analyzing the results, I noticed that we should have stripped `"faq-main"` from the filename on Day 1 when we were preparing the data. We should come back to it and adjust the ingestion process, but I won't do it here now.

We can also further adjust the instructions to make it cite the references immediately in the paragraph if we want.

Now we collect more data and finally start testing it.

LLM as a Judge

You can ask your colleagues to also do a "vibe check", but make sure you record the data. Often collecting 10-20 examples and manually inspecting them is enough to understand how your model is doing.

Don't be afraid of putting manual work into evaluation. Manual evaluation will help you understand edge cases, learn what good responses look like and think of evaluation criteria for automated checks later.

For example, I manually inspected the output and noticed that references are missing. So we will later add it as one of the checks.

So, in our case, we can have the following checks:

- Does the agent follow the instructions?
- Given the question, does the answer make sense?
- Does it include references?
- Did the agent use the available tools?

We don't have to evaluate this manually. Instead, we can delegate this to AI. This technique is called "LLM as a Judge".

The idea is simple: we use one LLM to evaluate the outputs of another LLM. This works because LLMs are good at following detailed evaluation criteria.

Our system prompt for the judge (we'll call it "evaluation agent" because it sounds cooler) can look like that:

```python
evaluation_prompt = """
Use this checklist to evaluate the quality of an AI agent's answer (<ANSWER>)
to a user question (<QUESTION>).
We also include the entire log (<LOG>) for analysis.
```

```
For each item, check if the condition is met.

Checklist:

- instructions_follow: The agent followed the user's instructions (in
<INSTRUCTIONS>)
- instructions_avoid: The agent avoided doing things it was told not to do
- answer_relevant: The response directly addresses the user's question
- answer_clear: The answer is clear and correct
- answer_citations: The response includes proper citations or sources when
required
- completeness: The response is complete and covers all key aspects of the
request
- tool_call_search: Is the search tool invoked?

Output true/false for each check and provide a short explanation for your
judgment.
""".strip()
```

Since we expect a very well defined structure of the response, we can use structured output.

We can define a Pydantic class with the expected response structure, and the LLM will produce output that matches this schema exactly.

This is how we do it:

Python

```python
from pydantic import BaseModel

class EvaluationCheck(BaseModel):
    check_name: str
    justification: str
    check_pass: bool

class EvaluationChecklist(BaseModel):
    checklist: list[EvaluationCheck]
    summary: str
```

This code defines the structure we expect from our evaluation:

- Each check has a name, justification, and pass/fail result

- The overall evaluation includes a list of checks and a summary

Note that `justification` comes before `check_pass`. This makes the LLM reason about the answer before giving the final judgment, which typically leads to better evaluation quality.

With Pydantic AI in order to make the output follow the specified class, we use the parameter `output_type`:

```python
eval_agent = Agent(
    name='eval_agent',
    model='gpt-5-nano',
    instructions=evaluation_prompt,
    output_type=EvaluationChecklist
)
```

Usually it's a good idea to evaluate the results of one model (in our case, "gpt-4o-mini") with another model (e.g. "gpt-5-nano"). A different model can catch mistakes, reduce self-bias, and give a second opinion. This makes evaluations more reliable.

We have the instructions, and we have the agent. In order to run the agent, it needs input. We'll start with a template:

```python
user_prompt_format = """
<INSTRUCTIONS>{instructions}</INSTRUCTIONS>
<QUESTION>{question}</QUESTION>
<ANSWER>{answer}</ANSWER>
<LOG>{log}</LOG>
""".strip()
```

We use XML markup because it's easier and more clear for LLMs to understand the input. XML tags help the model see the structure and boundaries of different sections in the prompt.

Let's fill it in. First, define a helper function for loading JSON log files:

```python
def load_log_file(log_file):
```

```
    with open(log_file, 'r') as f_in:
        log_data = json.load(f_in)
        log_data['log_file'] = log_file
        return log_data
```

We also add the filename in the result - it'll help us with tracking later.

Now let's use it:

```Python
log_record = load_log_file('./logs/faq_agent_v2_20250926_072928_467470.json')

instructions = log_record['system_prompt']
question = log_record['messages'][0]['parts'][0]['content']
answer = log_record['messages'][-1]['parts'][0]['content']
log = json.dumps(log_record['messages'])

user_prompt = user_prompt_format.format(
    instructions=instructions,
    question=question,
    answer=answer,
    log=log
)
```

The user input is ready and we can test it!

```Python
result = await eval_agent.run(user_prompt, output_type=EvaluationChecklist)

checklist = result.output
print(checklist.summary)

for check in checklist.checklist:
    print(check)
```

This code:

- Loads a saved interaction log
- Extracts the key components (instructions, question, answer, full log)
- Formats them into the evaluation prompt

- Runs the evaluation agent
- Prints the results

When we run it, we'll see this:

```
None
check_name='instructions_follow' justification='The assistant called the search
tool (log shows a text_search call) but did not follow the instruction to
include references (file names) from course materials; thus it did not fully
follow the provided instructions.' check_pass=False

check_name='instructions_avoid' justification='No forbidden actions in the
instructions were performed; the assistant did not do anything it was
explicitly told to avoid.' check_pass=True

check_name='answer_relevant' justification='The response directly addressed how
to install Kafka client libraries in Python and gave pip/conda commands, so it
is relevant to the question.' check_pass=True

check_name='answer_clear' justification='The answer is concise and the
installation commands are correct, though it omitted some caveats; overall it
is clear and understandable.' check_pass=True

check_name='answer_citations' justification='The reply did not include any
citations or filenames from the course materials as required by the
instructions.' check_pass=False

check_name='completeness' justification='The answer omitted important details
and alternatives (e.g., kafka-python library, librdkafka dependency, broker
setup, and course-material citations), so it is not fully complete.'
check_pass=False

check_name='tool_call_search' justification='The log shows a text_search tool
call was made before the answer was given.' check_pass=True
```

Note that we're putting the entire conversation log into the prompt, which is not really necessary. We can reduce it to make it less verbose.

For example, like that:

```python
def simplify_log_messages(messages):
    log_simplified = []

    for m in messages:
```

```
        parts = []

        for original_part in m['parts']:
            part = original_part.copy()
            kind = part['part_kind']

            if kind == 'user-prompt':
                del part['timestamp']
            if kind == 'tool-call':
                del part['tool_call_id']
            if kind == 'tool-return':
                del part['tool_call_id']
                del part['metadata']
                del part['timestamp']
                # Replace actual search results with placeholder to save tokens
                part['content'] = 'RETURN_RESULTS_REDACTED'
            if kind == 'text':
                del part['id']

            parts.append(part)

        message = {
            'kind': m['kind'],
            'parts': parts
        }

        log_simplified.append(message)
    return log_simplified
```

We make it simpler:

- remove timestamps and IDs that aren't needed for evaluation
- replace actual search results with a placeholder
- keep only the essential structure

This is helpful because it reduces the number of tokens we send to the evaluation model, which lowers the costs and speeds up evaluation.

Let's put everything together:

```
Python
async def evaluate_log_record(eval_agent, log_record):
    messages = log_record['messages']
```

```
    instructions = log_record['system_prompt']
    question = messages[0]['parts'][0]['content']
    answer = messages[-1]['parts'][0]['content']

    log_simplified = simplify_log_messages(messages)
    log = json.dumps(log_simplified)

    user_prompt = user_prompt_format.format(
        instructions=instructions,
        question=question,
        answer=answer,
        log=log
    )

    result = await eval_agent.run(user_prompt, output_type=EvaluationChecklist)
    return result.output


log_record = load_log_file('./logs/faq_agent_v2_20250926_072928_467470.json')
eval1 = await evaluate_log_record(eval_agent, log_record)
```

We know how to log our data and how to run evals on our logs.

Great. But how do we get more data to get a better understanding of the performance of our model?

Data Generation

We can ask AI to help. What if we used it for generating more questions? Let's do that.

We can sample some records from our database. Then for each record, ask an LLM to generate a question based on the record. We use this question as input to our agent and log the answers.

Let’s start by defining the question generator:

Python

```
question_generation_prompt = """
You are helping to create test questions for an AI agent that answers questions
about a data engineering course.
```

```
Based on the provided FAQ content, generate realistic questions that students
might ask.

The questions should:

- Be natural and varied in style
- Range from simple to complex
- Include both specific technical questions and general course questions

Generate one question for each record.
""".strip()

class QuestionsList(BaseModel):
    questions: list[str]

question_generator = Agent(
    name="question_generator",
    instructions=question_generation_prompt,
    model='gpt-4o-mini',
    output_type=QuestionsList
)
```

This prompt is designed for our specific use case (data engineering course FAQ). You should adjust it for your project.

We will send it a bunch of records, and it will generate a question from each of them.

Note: we use a simple way of generating questions. We can use a more complex approach where we also track the source (filename) of the question. If we do it, we can later check if this file was retrieved and cited in the answer. But we won't do it today to make things simpler.

Now let's sample 10 records from our dataset using Python's built-in `random.sample` function:

Python

```python
import random

sample = random.sample(de_dtc_faq, 10)
prompt_docs = [d['content'] for d in sample]
prompt = json.dumps(prompt_docs)

result = await question_generator.run(prompt)
```

```
questions = result.output.questions
```

Now we simply iterate over each of the question, ask our agent and log the results:

```python
from tqdm.auto import tqdm

for q in tqdm(questions):
    print(q)

    result = await agent.run(user_prompt=q)
    print(result.output)

    log_interaction_to_file(
        agent,
        result.new_messages(),
        source='ai-generated'
    )

    print()
```

We can repeat it multiple times until we have enough data. Around 100 should be good for a start, but today we can just continue with the 10 log records we already generated.

Using AI for generating test data is quite powerful. It can help us get data faster and sometimes cover edge cases we won't think about.

There are limitations too:

- AI-generated questions might not reflect real user behavior
- It may miss important edge cases that only real users encounter
- They may not capture the full complexity of real user queries

The logs are ready, so we can run evaluation on them with our evaluation agent.

First, collect all the AI-generated logs for the v2 agent:

```python
eval_set = []
```

```
for log_file in LOG_DIR.glob('*.json'):
    if 'faq_agent_v2' not in log_file.name:
        continue

    log_record = load_log_file(log_file)
    if log_record['source'] != 'ai-generated':
        continue

    eval_set.append(log_record)
```

And evaluate them:

```
Python
eval_results = []

for log_record in tqdm(eval_set):
    eval_result = await evaluate_log_record(eval_agent, log_record)
    eval_results.append((log_record, eval_result))
```

This code:

- Loops through each AI-generated log
- Runs our evaluation agent on it
- Stores both the original log and evaluation result

There are ways to speed this up, but we won't cover them in detail here. For example, you can try this:

- Don't ask for justification - this makes evaluation faster but slightly lower quality
- Parallelize execution - you can ask ChatGPT how to do this with async/await

The results are collected, but we need to display them and also calculate some statistics. The best tool for doing this is Pandas. We already should have it because minsearch depends on it.

But we can make it an explicit dependency:

```shell
uv add pandas
```

Our data is not ready to be converted to a Pandas DataFrame. We first need to transform it a little. Let's do it:

```python
rows = []

for log_record, eval_result in eval_results:
    messages = log_record['messages']

    row = {
        'file': log_record['log_file'].name,
        'question': messages[0]['parts'][0]['content'],
        'answer': messages[-1]['parts'][0]['content'],
    }

    checks = {c.check_name: c.check_pass for c in eval_result.checklist}
    row.update(checks)

    rows.append(row)
```

This code:

- Extracts key information from each log (file, question, answer)
- Converts the evaluation checks into a dictionary format

Now each row is a simple key-value dictionary, so we can create a DataFrame:

```python
import pandas as pd

df_evals = pd.DataFrame(rows)
```

We can look at individual records and see which checks are False.

But it's also useful to look at the overall stats:

```python
df_evals.mean(numeric_only=True)
```

This calculates the average pass rate for each check:

```
instructions_follow    0.3
instructions_avoid     1.0
answer_relevant        1.0
answer_clear           1.0
answer_citations       0.3
completeness           0.7
tool_call_search       1.0
```

This tells us:

- Only 30% of responses follow instructions completely
- All responses avoid forbidden actions (good!)
- All responses are relevant and clear (great!)
- Only 30% include proper citations (needs improvement)
- 70% of responses are complete
- All responses use the search tool (as expected)

For us, the most important check is `answer_relevant`. This tells us whether the agent actually answers the user's question. If this score was low, it'd mean that our agent is not ready.

We now know how to evaluate our agent. What can we do with it now?

Many things:

- Decide if this quality is good enough for deployment
- Evaluate different chunking approaches and search
- See if changing a prompt leads to any improvements.

The algorithm is simple:

- Collect data for evaluation and keep this dataset fixed
- Run different versions of your agent for this dataset
- Compare key metrics to decide which version is better

Evaluation is a very powerful tool and we should use it when possible.

Evaluating functions and tools

Also, we can (and should) evaluate our tools separately from evaluating the agent.

If it's code, we need to cover it with unit and integration tests.

We also have the search function, which we can evaluate using standard information retrieval metrics. For example:

- Precision and Recall: How many relevant results were retrieved vs. how many relevant results were missed
- Hit Rate: Percentage of queries that return at least one relevant result
- MRR (Mean Reciprocal Rank): Reflects the position of the first relevant result in the ranking

This is how we can implement hitrate and MRR calculation in Python:

```python
def evaluate_search_quality(search_function, test_queries):
    results = []

    for query, expected_docs in test_queries:
        search_results = search_function(query, num_results=5)

        # Calculate hit rate
        relevant_found = any(doc['filename'] in expected_docs for doc in
search_results)

        # Calculate MRR
        for i, doc in enumerate(search_results):
            if doc['filename'] in expected_docs:
                mrr = 1 / (i + 1)
                break
        else:
            mrr = 0

        results.append({
            'query': query,
            'hit': relevant_found,
            'mrr': mrr
        })
```

```
    return results
```

We won't do it today, but these ideas and the code will be useful when you implement a real agent project with search.

It's useful because it'll helps us make guided decisions about:

- When to use text vs. vector vs. hybrid search
- What are the best parameters for our search

You can ask ChatGPT to learn more about information retrieval evaluation metrics.

This was a very long lesson, but an important one. We finished it, and evaluated our agent. It’s good for deployment, so tomorrow we’ll create an UI for it and deploy it to the internet.

Homework

- Create an evaluation system for your agent
- Collect at least 10 interaction logs
- Set up automated evaluation using LLM as a judge
- Test different system prompts and compare results
- Make a post on social media about your evaluation process

Example post for LinkedIn

🧪 Day 5 of building AI agent: evaluation!

Today I learned how evals are crucial for AI agents:

• Built a logging system to capture all interactions
• Created automated evaluation using LLM as a judge
• Set up structured evaluation criteria
• Tested different system prompts

Key insight: start with "Vibe checks", collect enough data to evaluate your agent.

My project focuses on [YOUR PROJECT DESCRIPTION]. The evaluation system helps me compare different approaches and track improvements over time.

Here's my repo: [YOUR_REPO_LINK]

Next up: Building the interface!

Following along with this amazing course - who else is building AI agents?
You can sign up here: https://alexeygrigorev.com/aihero/

Example post for Twitter/X

Day 5: Built evals for my AI agent! 🧪

✅ Logging all interactions
✅ LLM-as-a-judge evaluation
✅ Structured evaluation criteria

We went from "vibe checks" to data-driven evals

Here's my repo: [YOUR_REPO_LINK]

Tomorrow: Building UI

Join me: https://alexeygrigorev.com/aihero/

Community

Have questions about this lesson or suggestions for improvement? You can find me and other learners in **DataTalks.Club Slack**:

- Join DataTalks.Club
- Find us in the `#course-ai-bootcamp` channel

In the community channel, you can:

- Ask questions about the course content
- Share your implementation and get feedback
- Show off your GitHub repositories
- Suggest improvements to the course materials
- Connect with other course participants

Don't hesitate to reach out - the community is here to help each other succeed!