Testing and Evals

With PydanticAI and LLM integrations in general, there are two distinct kinds of test:

1. **Unit tests** — tests of your application code, and whether it's behaving correctly
2. **Evals** — tests of the LLM, and how good or bad its responses are

For the most part, these two kinds of tests have pretty separate goals and considerations.

Unit tests

Unit tests for PydanticAI code are just like unit tests for any other Python code.

Because for the most part they're nothing new, we have pretty well established tools and patterns for writing and running these kinds of tests.

Unless you're really sure you know better, you'll probably want to follow roughly this strategy:

- Use `pytest` as your test harness
- If you find yourself typing out long assertions, use inline-snapshot
- Similarly, dirty-equals can be useful for comparing large data structures
- Use `TestModel` or `FunctionModel` in place of your actual model to avoid the cost, latency and variability of real LLM calls
- Use `Agent.override` to replace your model inside your application logic
- Set `ALLOW_MODEL_REQUESTS=False` globally to block any requests from being made to non-test models accidentally

Unit testing with `TestModel`

The simplest and fastest way to exercise most of your application code is using `TestModel`, this will (by default) call all tools in the agent, then return either plain text or a structured response depending on the return type of the agent.

`TestModel` is not magic

The "clever" (but not too clever) part of `TestModel` is that it will attempt to generate valid structured data for function tools and result types based on the schema of the registered tools.

There's no ML or AI in `TestModel`, it's just plain old procedural Python code that tries to generate data that satisfies the JSON schema of a tool.

The resulting data won't look pretty or relevant, but it should pass Pydantic's validation in most cases. If you want something more sophisticated, use `FunctionModel` and write your own data generation logic.

Let's write unit tests for the following application code:

weather_app.py

```python
import asyncio
from datetime import date

from pydantic_ai import Agent, RunContext

from fake_database import DatabaseConn  # (1)
from weather_service import WeatherService  # (2)

weather_agent = Agent(
    'openai:gpt-4o',
    deps_type=WeatherService,
    system_prompt='Providing a weather forecast at the locations the user provides.',
)


@weather_agent.tool
def weather_forecast(
    ctx: RunContext[WeatherService], location: str, forecast_date: date
) -> str:
    if forecast_date < date.today():  # (3)
        return ctx.deps.get_historic_weather(location, forecast_date)
    else:
        return ctx.deps.get_forecast(location, forecast_date)


async def run_weather_forecast(  # (4)
    user_prompts: list[tuple[str, int]], conn: DatabaseConn
):
    """Run weather forecast for a list of user prompts and save."""
    async with WeatherService() as weather_service:

        async def run_forecast(prompt: str, user_id: int):
            result = await weather_agent.run(prompt, deps=weather_service)
            await conn.store_forecast(user_id, result.data)

        # run all prompts in parallel
        await asyncio.gather(
            *(run_forecast(prompt, user_id) for (prompt, user_id) in user_prompts)
        )
```

1. `DatabaseConn` is a class that holds a database connection
2. `WeatherService` has methods to get weather forecasts and historic data about the weather
3. We need to call a different endpoint depending on whether the date is in the past or the future, you'll see why this nuance is important below
4. This function is the code we want to test, together with the agent it uses

Here we have a function that takes a list of `(user_prompt, user_id)` tuples, gets a weather forecast for each prompt, and stores the result in the database.

We want to test this code without having to mock certain objects or modify our code so we can pass test objects in.

Here's how we would write tests using `TestModel`:

test_weather_app.py

```python
from datetime import timezone
import pytest
```

```python
from dirty_equals import IsNow

from pydantic_ai import models
from pydantic_ai.models.test import TestModel
from pydantic_ai.messages import (
    SystemPrompt,
    UserPrompt,
    ModelStructuredResponse,
    ToolCall,
    ArgsDict,
    ToolReturn,
    ModelTextResponse,
)

from fake_database import DatabaseConn
from weather_app import run_weather_forecast, weather_agent

pytestmark = pytest.mark.anyio  # (1)
models.ALLOW_MODEL_REQUESTS = False  # (2)


async def test_forecast():
    conn = DatabaseConn()
    user_id = 1
    with weather_agent.override(model=TestModel()):  # (3)
        prompt = 'What will the weather be like in London on 2024-11-28?'
        await run_weather_forecast([(prompt, user_id)], conn)  # (4)

    forecast = await conn.get_forecast(user_id)
    assert forecast == '{"weather_forecast":"Sunny with a chance of rain"}'  # (5)

    assert weather_agent.last_run_messages == [  # (6)
        SystemPrompt(
            content='Providing a weather forecast at the locations the user provides.',
            role='system',
        ),
        UserPrompt(
            content='What will the weather be like in London on 2024-11-28?',
            timestamp=IsNow(tz=timezone.utc),  # (7)
            role='user',
        ),
        ModelStructuredResponse(
            calls=[
                ToolCall(
                    tool_name='weather_forecast',
                    args=ArgsDict(
                        args_dict={
                            'location': 'a',
                            'forecast_date': '2024-01-01',  # (8)
                        }
                    ),
                    tool_call_id=None,
                )
            ],
            timestamp=IsNow(tz=timezone.utc),
            role='model-structured-response',
        ),
        ToolReturn(
            tool_name='weather_forecast',
            content='Sunny with a chance of rain',
            tool_call_id=None,
            timestamp=IsNow(tz=timezone.utc),
            role='tool-return',
        ),
        ModelTextResponse(
            content='{"weather_forecast":"Sunny with a chance of rain"}',
            timestamp=IsNow(tz=timezone.utc),
            role='model-text-response',
        ),
    ]
```

1. We're using anyio to run async tests.
2. This is a safety measure to make sure we don't accidentally make real requests to the LLM while testing, see `ALLOW_MODEL_REQUESTS` for more details.
3. We're using `Agent.override` to replace the agent's model with `TestModel`, the nice thing about `override` is that we can replace the model inside agent without needing access to the agent `run*` methods call site.
4. Now we call the function we want to test inside the `override` context manager.
5. But default, `TestModel` will return a JSON string summarising the tools calls made, and what was returned. If you wanted to customise the response to something more closely aligned with the domain, you could add `custom_result_text='Sunny'` when defining `TestModel`.
6. So far we don't actually know which tools were called and with which values, we can use the `last_run_messages` attribute to inspect messages from the most recent run and assert the exchange between the agent and the model occurred as expected.
7. The `IsNow` helper allows us to use declarative asserts even with data which will contain timestamps that change over time.
8. `TestModel` isn't doing anything clever to extract values from the prompt, so these values are hardcoded.

Unit testing with `FunctionModel`

The above tests are a great start, but careful readers will notice that the `WeatherService.get_forecast` is never called since `TestModel` calls `weather_forecast` with a date in the past.

To fully exercise `weather_forecast`, we need to use `FunctionModel` to customise how the tools is called.

Here's an example of using `FunctionModel` to test the `weather_forecast` tool with custom inputs

test_weather_app2.py

```python
import re

import pytest

from pydantic_ai import models
from pydantic_ai.messages import (
    Message,
    ModelAnyResponse,
    ModelStructuredResponse,
    ModelTextResponse,
    ToolCall,
)
from pydantic_ai.models.function import AgentInfo, FunctionModel

from fake_database import DatabaseConn
from weather_app import run_weather_forecast, weather_agent

pytestmark = pytest.mark.anyio
models.ALLOW_MODEL_REQUESTS = False
```

```python
def call_weather_forecast(  # (1)
    messages: list[Message], info: AgentInfo
) -> ModelAnyResponse:
    if len(messages) == 2:
        # first call, call the weather forecast tool
        user_prompt = messages[1]
        m = re.search(r'\d{4}-\d{2}-\d{2}', user_prompt.content)
        assert m is not None
        args = {'location': 'London', 'forecast_date': m.group()}  # (2)
        return ModelStructuredResponse(
            calls=[ToolCall.from_dict('weather_forecast', args)]
        )
    else:
        # second call, return the forecast
        msg = messages[-1]
        assert msg.role == 'tool-return'
        return ModelTextResponse(f'The forecast is: {msg.content}')


async def test_forecast_future():
    conn = DatabaseConn()
    user_id = 1
    with weather_agent.override(model=FunctionModel(call_weather_forecast)):  # (3)
        prompt = 'What will the weather be like in London on 2032-01-01?'
        await run_weather_forecast([(prompt, user_id)], conn)

    forecast = await conn.get_forecast(user_id)
    assert forecast == 'The forecast is: Rainy with a chance of sun'
```

1. We define a function `call_weather_forecast` that will be called by `FunctionModel` in place of the LLM, this function has access to the list of `Message`s that make up the run, and `AgentInfo` which contains information about the agent and the function tools and return tools.
2. Our function is slightly intelligent in that it tries to extract a date from the prompt, but just hard codes the location.
3. We use `FunctionModel` to replace the agent's model with our custom function.

Overriding model via pytest fixtures

If you're writing lots of tests that all require model to be overridden, you can use pytest fixtures to override the model with `TestModel` or `FunctionModel` in a reusable way.

Here's an example of a fixture that overrides the model with `TestModel`:

tests.py

```python
import pytest
from weather_app import weather_agent

from pydantic_ai.models.test import TestModel


@pytest.fixture
def override_weather_agent():
    with weather_agent.override(model=TestModel()):
        yield


async def test_forecast(override_weather_agent: None):
    ...
    # test code here
```

Evals

"Evals" refers to evaluating a models performance for a specific application.

Warning

Unlike unit tests, evals are an emerging art/science; anyone who claims to know for sure exactly how your evals should be defined can safely be ignored.

Evals are generally more like benchmarks than unit tests, they never "pass" although they do "fail"; you care mostly about how they change over time.

Since evals need to be run against the real model, then can be slow and expensive to run, you generally won't want to run them in CI for every commit.

Measuring performance

The hardest part of evals is measuring how well the model has performed.

In some cases (e.g. an agent to generate SQL) there are simple, easy to run tests that can be used to measure performance (e.g. is the SQL valid? Does it return the right results? Does it return just the right results?).

In other cases (e.g. an agent that gives advice on quitting smoking) it can be very hard or impossible to make quantitative measures of performance — in the smoking case you'd really need to run a double-blind trial over months, then wait 40 years and observe health outcomes to know if changes to your prompt were an improvement.

There are a few different strategies you can use to measure performance:

- **End to end, self-contained tests** — like the SQL example, we can test the final result of the agent near-instantly
- **Synthetic self-contained tests** — writing unit test style checks that the output is as expected, checks like `'chewing gum' in response`, while these checks might seem simplistic they can be helpful, one nice characteristic is that it's easy to tell what's wrong when they fail
- **LLMs evaluating LLMs** — using another models, or even the same model with a different prompt to evaluate the performance of the agent (like when the class marks each other's homework because the teacher has a hangover), while the downsides and complexities of this approach are obvious, some think it can be a useful tool in the right circumstances
- **Evals in prod** — measuring the end results of the agent in production, then creating a quantitative measure of performance, so you can easily measure changes over time as you change the prompt or model used, logfire can be extremely useful in this case since you can write a custom query to measure the performance of your agent

System prompt customization

The system prompt is the developer's primary tool in controlling an agent's behavior, so it's often useful to be able to customise the system prompt and see how performance changes. This is particularly relevant when the system prompt contains a list of examples and you want to understand how changing that list affects the model's performance.

Let's assume we have the following app for running SQL generated from a user prompt (this examples omits a lot of details for brevity, see the SQL gen example for a more complete code):

sql_app.py

```python
import json
from pathlib import Path
```

```python
from typing import Union

from pydantic_ai import Agent, RunContext

from fake_database import DatabaseConn


class SqlSystemPrompt:  # (1)
    def __init__(
        self, examples: Union[list[dict[str, str]], None] = None, db: str = 'PostgreSQL'
    ):
        if examples is None:
            # if examples aren't provided, load them from file, this is the default
            with Path('examples.json').open('rb') as f:
                self.examples = json.load(f)
        else:
            self.examples = examples

        self.db = db

    def build_prompt(self) -> str:  # (2)
        return f"""\
Given the following {self.db} table of records, your job is to
write a SQL query that suits the user's request.

Database schema:
CREATE TABLE records (
  ...
);

{''.join(self.format_example(example) for example in self.examples)}
"""

    @staticmethod
    def format_example(example: dict[str, str]) -> str:  # (3)
        return f"""\
<example>
  <request>{example['request']}</request>
  <sql>{example['sql']}</sql>
</example>
"""


sql_agent = Agent(
    'gemini-1.5-flash',
    deps_type=SqlSystemPrompt,
)


@sql_agent.system_prompt
async def system_prompt(ctx: RunContext[SqlSystemPrompt]) -> str:
    return ctx.deps.build_prompt()


async def user_search(user_prompt: str) -> list[dict[str, str]]:
    """Search the database based on the user's prompts."""
    ...  # (4)
    result = await sql_agent.run(user_prompt, deps=SqlSystemPrompt())
    conn = DatabaseConn()
    return await conn.execute(result.data)
```

1. The `SqlSystemPrompt` class is used to build the system prompt, it can be customised with a list of examples and a database type. We implement this as a separate class passed as a dep to the agent so we can override both the inputs and the logic during evals via dependency injection.
2. The `build_prompt` method constructs the system prompt from the examples and the database type.
3. Some people think that LLMs are more likely to generate good responses if examples are formatted as XML as it's to identify the end of a string, see #93.
4. In reality, you would have more logic here, making it impractical to run the agent independently of the wider application.

`examples.json` looks something like this:

```
request: show me error records with the tag "foobar"
response: SELECT * FROM records WHERE level = 'error' and 'foobar' = ANY(tags)
```

examples.json

```json
{
  "examples": [
    {
      "request": "Show me all records",
      "sql": "SELECT * FROM records;"
    },
    {
      "request": "Show me all records from 2021",
      "sql": "SELECT * FROM records WHERE date_trunc('year', date) = '2021-01-01';"
    },
    {
      "request": "show me error records with the tag 'foobar'",
      "sql": "SELECT * FROM records WHERE level = 'error' and 'foobar' = ANY(tags);"
    },
    ...
  ]
}
```

Now we want a way to quantify the success of the SQL generation so we can judge how changes to the agent affect its performance.

We can use `Agent.override` to replace the system prompt with a custom one that uses a subset of examples, and then run the application code (in this case `user_search`). We also run the actual SQL from the examples and compare the "correct" result from the example SQL to the SQL generated by the agent. (We compare the results of running the SQL rather than the SQL itself since the SQL might be semantically equivalent but written in a different way).

To get a quantitative measure of performance, we assign points to each run as follows: * **-100** points if the generated SQL is invalid * **-1** point for each row returned by the agent (so returning lots of results is discouraged) * **+5** points for each row returned by the agent that matches the expected result

We use 5-fold cross-validation to judge the performance of the agent using our existing set of examples.

sql_app_evals.py

```python
import json
import statistics
from pathlib import Path
from itertools import chain

from fake_database import DatabaseConn, QueryError
from sql_app import sql_agent, SqlSystemPrompt, user_search


async def main():
```

```python
with Path('examples.json').open('rb') as f:
    examples = json.load(f)

# split examples into 5 folds
fold_size = len(examples) // 5
folds = [examples[i : i + fold_size] for i in range(0, len(examples), fold_size)]
conn = DatabaseConn()
scores = []

for i, fold in enumerate(folds, start=1):
    fold_score = 0
    # build all other folds into a list of examples
    other_folds = list(chain(*(f for j, f in enumerate(folds) if j != i)))
    # create a new system prompt with the other fold examples
    system_prompt = SqlSystemPrompt(examples=other_folds)

    # override the system prompt with the new one
    with sql_agent.override(deps=system_prompt):
        for case in fold:
            try:
                agent_results = await user_search(case['request'])
            except QueryError as e:
                print(f'Fold {i} {case}: {e}')
                fold_score -= 100
            else:
                # get the expected results using the SQL from this case
                expected_results = await conn.execute(case['sql'])

                agent_ids = [r['id'] for r in agent_results]
                # each returned value has a score of -1
                fold_score -= len(agent_ids)
                expected_ids = {r['id'] for r in expected_results}

                # each return value that matches the expected value has a score of 3
                fold_score += 5 * len(set(agent_ids) & expected_ids)

    scores.append(fold_score)

overall_score = statistics.mean(scores)
print(f'Overall score: {overall_score:0.2f}')
#> Overall score: 12.00
```

We can then change the prompt, the model, or the examples and see how the score changes over time.