Pydantic Documentation

Generated on: 2024-12-12 12:37:42

==
Page: PydanticAI
URL: https://ai.pydantic.dev/
==

Introduction
Agent Framework / shim to use Pydantic with LLMs
When I first found FastAPI, I got it immediately. I was excited to find something so innovative and ergonomic built on Pydantic.
Virtually every Agent Framework and LLM library in Python uses Pydantic, but when we began to use

LLMs in
Pydantic Logfire
, I couldn't find anything that gave me the same feeling.
PydanticAI is a Python Agent Framework designed to make it less painful to build production grade applications with Generative AI.
Why use PydanticAI
Built by the team behind Pydantic (the validation layer of the OpenAI SDK, the Anthropic SDK, LangChain, LlamaIndex, AutoGPT, Transformers, CrewAI, Instructor and many more)
Model-agnostic — currently OpenAI, Gemini, Anthropic, and Groq are supported, Anthropic
is coming soon
. And there is a simple interface to implement support for other models.
Type-safe
Control flow and agent composition is done with vanilla Python, allowing you to make use of the same Python development best practices you'd use in any other (non-AI) project
Structured response
validation with Pydantic
Streamed responses
, including validation of streamed
structured
responses with Pydantic
Novel, type-safe
dependency injection system
, useful for testing and eval-driven iterative development
Logfire integration
for debugging and monitoring the performance and general behavior of your LLM-powered application
In Beta
PydanticAI is in early beta, the API is still subject to change and there's a lot more to do.
Feedback
is very welcome!
Hello World Example
Here's a minimal example of PydanticAI:
hello_world.py

```
from
pydantic_ai
import
Agent
agent
=
Agent
(
# (1)!
'gemini-1.5-flash'
,
system_prompt
=
'Be concise, reply with one sentence.'
,
# (2)!
)
result
=
agent
.
run_sync
(
'Where does "hello world" come from?'
)
# (3)!
print
(
result
.
data
)
"""
The first known use of "hello, world" was in a 1974 textbook about the C programming language.
"""
```

Define a very simple agent — here we configure the agent to use
Gemini 1.5's Flash
model, but you can also set the model when running the agent.
Register a static
system prompt
using a keyword argument to the agent. For more complex dynamically-generated system prompts, see the example below.
Run the agent
synchronously, conducting a conversation with the LLM. Here the exchange should be very short: PydanticAI will send the system prompt and the user query to the LLM, the model will return a text

response.
(This example is complete, it can be run "as is")
Not very interesting yet, but we can easily add "tools", dynamic system prompts, and structured responses to build more powerful agents.
Tools & Dependency Injection Example
Here is a concise example using PydanticAI to build a support agent for a bank:
bank_support.py

```
from
dataclasses
import
dataclass
from
pydantic
import
BaseModel
,
Field
from
pydantic_ai
import
Agent
,
RunContext
from
bank_database
import
DatabaseConn
@dataclass
class
SupportDependencies
:
# (3)!
customer_id
:
int
db
:
DatabaseConn
# (12)!
class
SupportResult
(
BaseModel
):
# (13)!
support_advice
:
str
=
Field
(
description
=
'Advice returned to the customer'
)
block_card
:
bool
=
Field
(
description
=
"Whether to block the customer's card"
)
risk
:
int
=
Field
(
description
=
'Risk level of query'
,
ge
=
```

```
0
,
le
=
10
)
support_agent
=
Agent
(
# (1)!
'openai:gpt-4o'
,
# (2)!
deps_type
=
SupportDependencies
,
result_type
=
SupportResult
,
# (9)!
system_prompt
=
(
# (4)!
'You are a support agent in our bank, give the '
'customer support and judge the risk level of their query.'
),
)
@support_agent
.
system_prompt
# (5)!
async
def
add_customer_name
(
ctx
:
RunContext
[
SupportDependencies
])
->
str
:
customer_name
=
await
ctx
.
deps
.
db
.
customer_name
(
id
=
ctx
.
deps
.
customer_id
)
return
f
"The customer's name is
{
customer_name
!r}
"
@support_agent
.
tool
```

```
# (6)!
async
def
customer_balance
(
ctx
:
RunContext
[
SupportDependencies
],
include_pending
:
bool
)
->
float
:
"""Returns the customer's current account balance."""
# (7)!
return
await
ctx
.
deps
.
db
.
customer_balance
(
id
=
ctx
.
deps
.
customer_id
,
include_pending
=
include_pending
,
)
...
# (11)!
async
def
main
():
deps
=
SupportDependencies
(
customer_id
=
123
,
db
=
DatabaseConn
())
result
=
await
support_agent
.
run
(
'What is my balance?'
,
deps
=
deps
)
# (8)!
print
(
```

result
.
data
)
(10)!
"""
support_advice='Hello John, your current account balance, including pending transactions, is $123.45.' block_card=False risk=1
"""
result
=
await
support_agent
.
run
(
'I just lost my card!'
,
deps
=
deps
)
print
(
result
.
data
)
"""
support_advice="I'm sorry to hear that, John. We are temporarily blocking your card to prevent unauthorized transactions." block_card=True risk=8
"""
This
agent
will act as first-tier support in a bank. Agents are generic in the type of dependencies they accept and the type of result they return. In this case, the support agent has type
Agent
[
SupportDependencies
,
SupportResult
]
.
Here we configure the agent to use
OpenAI's GPT-4o model
, you can also set the model when running the agent.
The
SupportDependencies
dataclass is used to pass data, connections, and logic into the model that will be needed when running
system prompt
and
tool
functions. PydanticAI's system of dependency injection provides a
type-safe
way to customise the behavior of your agents, and can be especially useful when running
unit tests
and evals.
Static
system prompts
can be registered with the
system_prompt
keyword argument
to the agent.
Dynamic
system prompts
can be registered with the
@agent.system_prompt
decorator, and can make use of dependency injection. Dependencies are carried via the
RunContext
argument, which is parameterized with the
deps_type
from above. If the type annotation here is wrong, static type checkers will catch it.
tool
let you register functions which the LLM may call while responding to a user. Again, dependencies are carried via
RunContext

, any other arguments become the tool schema passed to the LLM. Pydantic is used to validate these arguments, and errors are passed back to the LLM so it can retry.
The docstring of a tool is also passed to the LLM as the description of the tool. Parameter descriptions are
extracted
from the docstring and added to the parameter schema sent to the LLM.
Run the agent
asynchronously, conducting a conversation with the LLM until a final response is reached. Even in this fairly simple case, the agent will exchange multiple messages with the LLM as tools are called to retrieve a result.
The response from the agent will, be guaranteed to be a
SupportResult
, if validation fails
reflection
will mean the agent is prompted to try again.
The result will be validated with Pydantic to guarantee it is a
SupportResult
, since the agent is generic, it'll also be typed as a
SupportResult
to aid with static type checking.
In a real use case, you'd add more tools and a longer system prompt to the agent to extend the context it's equipped with and support it can provide.
This is a simple sketch of a database connection, used to keep the example short and readable. In reality, you'd be connecting to an external database (e.g. PostgreSQL) to get information about customers.
This
Pydantic
model is used to constrain the structured data returned by the agent. From this simple definition, Pydantic builds the JSON Schema that tells the LLM how to return the data, and performs validation to guarantee the data is correct at the end of the run.
Complete
bank_support.py
example
The code included here is incomplete for the sake of brevity (the definition of
DatabaseConn
is missing); you can find the complete
bank_support.py
example
here
.
Instrumentation with Pydantic Logfire
To understand the flow of the above runs, we can watch the agent in action using Pydantic Logfire.
To do this, we need to set up logfire, and add the following to our code:
bank_support_with_logfire.py

```
...
from
bank_database
import
DatabaseConn
import
logfire
logfire
.
configure
()
# (1)!
logfire
.
instrument_asyncpg
()
# (2)!
...
```

Configure logfire, this will fail if not project is set up.
In our demo,
DatabaseConn
uses
asyncpg
to connect to a PostgreSQL database, so
logfire.instrument_asyncpg()
is used to log the database queries.
That's enough to get the following view of your agent in action:
See
Monitoring and Performance
to learn more.
Next Steps
To try PydanticAI yourself, follow the instructions
in the examples

.
Read the
docs
to learn more about building applications with PydanticAI.
Read the
API Reference
to understand PydanticAI's interface.

==
Page: Pydantic Model - PydanticAI
URL: https://ai.pydantic.dev/examples/pydantic-model/
==

Pydantic Model - PydanticAI
Skip to content
PydanticAI
Pydantic Model
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

Introduction
Examples
Pydantic Model
Simple example of using PydanticAI to construct a Pydantic model from a text input.
Demonstrates:
structured
result_type
Running the Example
With

dependencies installed and environment variables set
, run:
pip
uv
python
-m
pydantic_ai_examples.pydantic_model
uv
run
-m
pydantic_ai_examples.pydantic_model
This examples uses
openai:gpt-4o
by default, but it works well with other models, e.g. you can run it
with Gemini using:
pip
uv
PYDANTIC_AI_MODEL
=
gemini-1.5-pro
python
-m
pydantic_ai_examples.pydantic_model
PYDANTIC_AI_MODEL
=
gemini-1.5-pro
uv
run
-m
pydantic_ai_examples.pydantic_model
(or
PYDANTIC_AI_MODEL=gemini-1.5-flash ...
)
Example Code
pydantic_model.py
import
os
from
typing
import
cast
import
logfire
from
pydantic
import
BaseModel
from
pydantic_ai
import
Agent
from
pydantic_ai.models
import
KnownModelName
'if-token-present' means nothing will be sent (and the example will work) if you don't have
logfire configured
logfire
.
configure
(
send_to_logfire
=
'if-token-present'
)
class
MyModel
(
BaseModel
):
city
:
str
country
:
str
model

```
=
cast
(
KnownModelName
,
os
.
getenv
(
'PYDANTIC_AI_MODEL'
,
'openai:gpt-4o'
))
print
(
f
'Using model:
{
model
}
'
)
agent
=
Agent
(
model
,
result_type
=
MyModel
)
if
__name__
==
'__main__'
:
result
=
agent
.
run_sync
(
'The windy city in the US of A.'
)
print
(
result
.
data
)
print
(
result
.
cost
())
```


==
Page: pydantic_ai.exceptions - PydanticAI
URL: https://ai.pydantic.dev/api/exceptions/
==

pydantic_ai.exceptions - PydanticAI

Table of contents
exceptions
ModelRetry
message
UserError
message
UnexpectedModelBehavior
message
body
Introduction
API Reference
pydantic_ai.exceptions
ModelRetry
Bases:
Exception
Exception raised when a tool function should be retried.
The agent will return the message to the model and ask it to try calling the function/tool again.
Source code in
pydantic_ai_slim/pydantic_ai/exceptions.py

```
class
ModelRetry
```

```
(
Exception
):
"""Exception raised when a tool function should be retried.
The agent will return the message to the model and ask it to try calling the function/tool again.
"""
message
:
str
"""The message to return to the model."""
def
__init__
(
self
,
message
:
str
):
self
.
message
=
message
super
()
.
__init__
(
message
)
```

message
instance-attribute

```
message
:
str
=
message
```

The message to return to the model.

UserError

Bases:
RuntimeError

Error caused by a usage mistake by the application developer — You!

Source code in
pydantic_ai_slim/pydantic_ai/exceptions.py

```
class
UserError
(
RuntimeError
):
"""Error caused by a usage mistake by the application developer — You!"""
message
:
str
"""Description of the mistake."""
def
__init__
(
self
,
message
:
str
):
self
.
message
=
```

```
message
super
()
.
__init__
(
message
)
```
message
instance-attribute
```
message
:
str
=
message
```
Description of the mistake.

UnexpectedModelBehavior

Bases:
RuntimeError

Error caused by unexpected Model behavior, e.g. an unexpected response code.

Source code in
pydantic_ai_slim/pydantic_ai/exceptions.py
```
class
UnexpectedModelBehavior
(
RuntimeError
):
"""Error caused by unexpected Model behavior, e.g. an unexpected response code."""
message
:
str
"""Description of the unexpected behavior."""
body
:
str
|
None
"""The body of the response, if available."""
def
__init__
(
self
,
message
:
str
,
body
:
str
|
None
=
```

```
None
):
self
.
message
=
message
if
body
is
None
:
self
.
body
:
str
|
None
=
None
else
:
try
:
self
.
body
=
json
.
dumps
(
json
.
loads
(
body
),
indent
=
2
)
except
ValueError
:
self
.
body
=
body
super
()
.
__init__
(
message
)
def
__str__
(
self
)
->
str
:
if
self
.
body
:
return
f
'
{
self
.
```

message
}
, body:
\n
{
self
.
body
}
'
else
:
return
self
.
message
message
instance-attribute
message
:
str
=
message
Description of the unexpected behavior.
body
instance-attribute
body
:
str
|
None
=
dumps
(
loads
(
body
),
indent
=
2
)
The body of the response, if available.

==
Page: Debugging and Monitoring - PydanticAI
URL: https://ai.pydantic.dev/logfire/
==

Debugging and Monitoring - PydanticAI

Debugging and Monitoring
Applications that use LLMs have some challenges that are well known and understood: LLMs are
slow
,
unreliable
and
expensive
.
These applications also have some challenges that most developers have encountered much less often:
LLMs are
fickle
and
non-deterministic
. Subtle changes in a prompt can completely change a model's performance, and there's no
EXPLAIN
query you can run to understand why.
Warning
From a software engineers point of view, you can think of LLMs as the worst database you've ever heard of, but worse.
If LLMs weren't so bloody useful, we'd never touch them.
To build successful applications with LLMs, we need new tools to understand both model performance, and the behavior of applications that rely on them.
LLM Observability tools that just let you understand how your model is performing are useless: making API calls to an LLM is easy, it's building that into an application that's hard.
Pydantic Logfire
Pydantic Logfire
is an observability platform developed by the team who created and maintain Pydantic and PydanticAI.
Logfire aims to let you understand your entire application: Gen AI, classic predictive AI, HTTP traffic, database queries and everything else a modern application needs.
Pydantic Logfire is a commercial product
Logfire is a commercially supported, hosted platform with an extremely generous and perpetual
free tier
.
You can sign up and start using Logfire in a couple of minutes.
PydanticAI has built-in (but optional) support for Logfire via the
logfire-api
no-op package.
That means if the
logfire
package is installed and configured, detailed information about agent runs is sent to Logfire. But
if the
logfire
package is not installed, there's virtually no overhead and nothing is sent.
Here's an example showing details of running the

Weather Agent
in Logfire:
Using Logfire
To use logfire, you'll need a logfire
account
, and logfire installed:
pip
uv
pip
install
'pydantic-ai[logfire]'
uv
add
'pydantic-ai[logfire]'
Then authenticate your local environment with logfire:
pip
uv
logfire
auth
uv
run
logfire
auth
And configure a project to send data to:
pip
uv
logfire
projects
new
uv
run
logfire
projects
new
(Or use an existing project with
logfire projects use
)
The last step is to add logfire to your code:
adding_logfire.py
import
logfire
logfire
.
configure
()
The
logfire documentation
has more details on how to use logfire, including how to instrument other libraries like Pydantic, HTTPX and FastAPI.
Since Logfire is build on
OpenTelemetry
, you can use the Logfire Python SDK to send data to any OpenTelemetry collector.
Once you have logfire set up, there are two primary ways it can help you understand your application:
Debugging
— Using the live view to see what's happening in your application in real-time.
Monitoring
— Using SQL and dashboards to observe the behavior of your application, Logfire is effectively a SQL database that stores information about how your application is running.
Debugging
To demonstrate how Logfire can let you visualise the flow of a PydanticAI run, here's the view you get from Logfire while running the
chat app examples
:
Monitoring Performance
We can also query data with SQL in Logfire to monitor the performance of an application. Here's a real world example of using Logfire to monitor PydanticAI runs inside Logfire itself:

==
Page: pydantic_ai.models.openai - PydanticAI
URL: https://ai.pydantic.dev/api/models/openai/
==

pydantic_ai.models.openai - PydanticAI
Skip to content
PydanticAI

pydantic_ai.models.openai
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai
Introduction
Installation & Setup
Getting Help
Contributing
Documentation
Documentation
Agents
Dependencies
Results
Messages and chat history
Testing and Evals
Debugging and Monitoring
Examples
Examples
Pydantic Model
Weather agent
Bank support
SQL Generation
RAG
Stream markdown
Stream whales
Chat App with FastAPI
API Reference
API Reference
pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.openai
Table of contents
Setup
openai
OpenAIModelName
OpenAIModel
__init__
OpenAIAgentModel
OpenAIStreamTextResponse
OpenAIStreamStructuredResponse
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Setup
openai
OpenAIModelName
OpenAIModel
__init__
OpenAIAgentModel
OpenAIStreamTextResponse
OpenAIStreamStructuredResponse
Introduction
API Reference
pydantic_ai.models.openai
Setup
For details on how to set up authentication with this model, see
model configuration for OpenAI
.
OpenAIModelName
module-attribute
OpenAIModelName
=
Union
[
ChatModel
,

str
]
Using this more broad type for the model name instead of the ChatModel definition
allows this model to be used more easily with other model types (ie, Ollama)
OpenAIModel
dataclass
Bases:
Model
A model that uses the OpenAI API.
Internally, this uses the
OpenAI Python client
to interact with the API.
Apart from
__init__
, all methods are private or match those of the base class.
Source code in
pydantic_ai_slim/pydantic_ai/models/openai.py

```
@dataclass
(
init
=
False
)
class
OpenAIModel
(
Model
):
"""A model that uses the OpenAI API.
Internally, this uses the [OpenAI Python client](https://github.com/openai/openai-python) to
interact with the API.
Apart from `__init__`, all methods are private or match those of the base class.
"""
model_name
:
OpenAIModelName
client
:
AsyncOpenAI
=
field
(
repr
=
False
)
def
__init__
(
self
,
model_name
:
OpenAIModelName
,
*
,
api_key
:
str
|
None
=
None
,
openai_client
:
AsyncOpenAI
|
None
=
None
,
http_client
:
AsyncHTTPClient
|
None
=
None
```

```
,
):
"""Initialize an OpenAI model.
Args:
model_name: The name of the OpenAI model to use. List of model names available
[here](https://github.com/openai/openai-python/blob/v1.54.3/src/openai/types/chat_model.py#L7)
(Unfortunately, despite being ask to do so, OpenAI do not provide `.inv` files for their API).
api_key: The API key to use for authentication, if not provided, the `OPENAI_API_KEY` environment
variable
will be used if available.
openai_client: An existing
[`AsyncOpenAI`](https://github.com/openai/openai-python?tab=readme-ov-file#async-usage)
client to use, if provided, `api_key` and `http_client` must be `None`.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
"""
self
.
model_name
:
OpenAIModelName
=
model_name
if
openai_client
is
not
None
:
assert
http_client
is
None
,
'Cannot provide both `openai_client` and `http_client`'
assert
api_key
is
None
,
'Cannot provide both `openai_client` and `api_key`'
self
.
client
=
openai_client
elif
http_client
is
not
None
:
self
.
client
=
AsyncOpenAI
(
api_key
=
api_key
,
http_client
=
http_client
)
else
:
self
.
client
=
AsyncOpenAI
(
api_key
=
api_key
,
```

```
http_client
=
cached_async_http_client
())
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
],
)
->
AgentModel
:
check_allow_model_requests
()
tools
=
[
self
.
_map_tool_definition
(
r
)
for
r
in
function_tools
]
if
result_tools
:
tools
+=
[
self
.
_map_tool_definition
(
r
)
for
r
in
result_tools
]
return
OpenAIAgentModel
(
self
.
client
,
self
.
model_name
,
allow_text_result
```

```
,
tools
,
)
def
name
(
self
)
->
str
:
return
f
'openai:
{
self
.
model_name
}
'
@staticmethod
def
_map_tool_definition
(
f
:
ToolDefinition
)
->
chat
.
ChatCompletionToolParam
:
return
{
'type'
:
'function'
,
'function'
:
{
'name'
:
f
.
name
,
'description'
:
f
.
description
,
'parameters'
:
f
.
parameters_json_schema
,
},
}
__init__
__init__
(
model_name
:
OpenAIModelName
,
*
,
api_key
:
str
|
None
```

=
None
,
openai_client
:
AsyncOpenAI
|
None
=
None
,
http_client
:
AsyncClient
|
None
=
None
)
Initialize an OpenAI model.
Parameters:
Name
Type
Description
Default
model_name
OpenAIModelName
The name of the OpenAI model to use. List of model names available
here
(Unfortunately, despite being ask to do so, OpenAI do not provide
.inv
files for their API).
required
api_key
str
| None
The API key to use for authentication, if not provided, the
OPENAI_API_KEY
environment variable
will be used if available.
None
openai_client
AsyncOpenAI
| None
An existing
AsyncOpenAI
client to use, if provided,
api_key
and
http_client
must be
None
.
None
http_client
AsyncClient
| None
An existing
httpx.AsyncClient
to use for making HTTP requests.
None
Source code in
pydantic_ai_slim/pydantic_ai/models/openai.py

```
def
__init__
(
self
,
model_name
:
OpenAIModelName
,
*
,
api_key
:
str
|
None
=
None
,
openai_client
:
AsyncOpenAI
|
None
=
None
,
http_client
:
AsyncHTTPClient
|
None
=
None
,
):
"""Initialize an OpenAI model.
Args:
model_name: The name of the OpenAI model to use. List of model names available
[here](https://github.com/openai/openai-python/blob/v1.54.3/src/openai/types/chat_model.py#L7)
(Unfortunately, despite being ask to do so, OpenAI do not provide `.inv` files for their API).
api_key: The API key to use for authentication, if not provided, the `OPENAI_API_KEY` environment
variable
will be used if available.
openai_client: An existing
[`AsyncOpenAI`](https://github.com/openai/openai-python?tab=readme-ov-file#async-usage)
client to use, if provided, `api_key` and `http_client` must be `None`.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
"""
self
.
model_name
:
OpenAIModelName
=
model_name
if
openai_client
is
not
None
```

```
:
assert
http_client
is
None
,
'Cannot provide both `openai_client` and `http_client`'
assert
api_key
is
None
,
'Cannot provide both `openai_client` and `api_key`'
self
.
client
=
openai_client
elif
http_client
is
not
None
:
self
.
client
=
AsyncOpenAI
(
api_key
=
api_key
,
http_client
=
http_client
)
else
:
self
.
client
=
AsyncOpenAI
(
api_key
=
api_key
,
http_client
=
cached_async_http_client
())
```

OpenAIAgentModel
dataclass
Bases:
AgentModel
Implementation of
AgentModel
for OpenAI models.
Source code in
pydantic_ai_slim/pydantic_ai/models/openai.py

```
@dataclass
class OpenAIAgentModel(AgentModel):
"""Implementation of `AgentModel` for OpenAI models."""
client: AsyncOpenAI
model_name: OpenAIModelName
allow_text_result: bool
tools: list[chat.ChatCompletionToolParam]
async def request(self, messages: list[Message])
```

```
->
tuple
[
ModelAnyResponse
,
result
.
Cost
]:
response
=
await
self
.
_completions_create
(
messages
,
False
)
return
self
.
_process_response
(
response
),
_map_cost
(
response
)
@asynccontextmanager
async
def
request_stream
(
self
,
messages
:
list
[
Message
])
->
AsyncIterator
[
EitherStreamedResponse
]:
response
=
await
self
.
_completions_create
(
messages
,
True
)
async
with
response
:
yield
await
self
.
_process_streamed_response
(
response
)
@overload
async
def
_completions_create
(
```

```
self
,
messages
:
list
[
Message
],
stream
:
Literal
[
True
]
)
->
AsyncStream
[
ChatCompletionChunk
]:
pass
@overload
async
def
_completions_create
(
self
,
messages
:
list
[
Message
],
stream
:
Literal
[
False
])
->
chat
.
ChatCompletion
:
pass
async
def
_completions_create
(
self
,
messages
:
list
[
Message
],
stream
:
bool
)
->
chat
.
ChatCompletion
|
AsyncStream
[
ChatCompletionChunk
]:
# standalone function to make it easier to override
if
not
self
.
tools
```

```
:
tool_choice
:
Literal
[
'none'
,
'required'
,
'auto'
]
|
None
=
None
elif
not
self
.
allow_text_result
:
tool_choice
=
'required'
else
:
tool_choice
=
'auto'
openai_messages
=
[
self
.
_map_message
(
m
)
for
m
in
messages
]
return
await
self
.
client
.
chat
.
completions
.
create
(
model
=
self
.
model_name
,
messages
=
openai_messages
,
n
=
1
,
parallel_tool_calls
=
True
if
self
.
tools
else
```

```
NOT_GIVEN
,
tools
=
self
.
tools
or
NOT_GIVEN
,
tool_choice
=
tool_choice
or
NOT_GIVEN
,
stream
=
stream
,
stream_options
=
{
'include_usage'
:
True
}
if
stream
else
NOT_GIVEN
,
)
@staticmethod
def
_process_response
(
response
:
chat
.
ChatCompletion
)
->
ModelAnyResponse
:
"""Process a non-streamed response, and prepare a message to return."""
timestamp
=
datetime
.
fromtimestamp
(
response
.
created
,
tz
=
timezone
.
utc
)
choice
=
response
.
choices
[
0
]
if
choice
.
message
.
tool_calls
```

```
is
not
None
:
return
ModelStructuredResponse
(
[
ToolCall
.
from_json
(
c
.
function
.
name
,
c
.
function
.
arguments
,
c
.
id
)
for
c
in
choice
.
message
.
tool_calls
],
timestamp
=
timestamp
,
)
else
:
assert
choice
.
message
.
content
is
not
None
,
choice
return
ModelTextResponse
(
choice
.
message
.
content
,
timestamp
=
timestamp
)
@staticmethod
async
def
_process_streamed_response
(
response
:
AsyncStream
[
```

```
ChatCompletionChunk
])
->
EitherStreamedResponse
:
"""Process a streamed response, and prepare a streaming response to return."""
timestamp
:
datetime
|
None
=
None
start_cost
=
Cost
()
# the first chunk may contain enough information so we iterate until we get either `tool_calls` or
`content`
while
True
:
try
:
chunk
=
await
response
.
__anext__
()
except
StopAsyncIteration
as
e
:
raise
UnexpectedModelBehavior
(
'Streamed response ended without content or tool calls'
)
from
e
timestamp
=
timestamp
or
datetime
.
fromtimestamp
(
chunk
.
created
,
tz
=
timezone
.
utc
)
start_cost
+=
_map_cost
(
chunk
)
if
chunk
.
choices
:
delta
=
chunk
.
choices
```

```
[
0
]
.
delta
if
delta
.
content
is
not
None
:
return
OpenAIStreamTextResponse
(
delta
.
content
,
response
,
timestamp
,
start_cost
)
elif
delta
.
tool_calls
is
not
None
:
return
OpenAIStreamStructuredResponse
(
response
,
{
c
.
index
:
c
for
c
in
delta
.
tool_calls
},
timestamp
,
start_cost
,
)
# else continue until we get either delta.content or delta.tool_calls
@staticmethod
def
_map_message
(
message
:
Message
)
->
chat
.
ChatCompletionMessageParam
:
"""Just maps a `pydantic_ai.Message` to a `openai.types.ChatCompletionMessageParam`."""
if
message
.
role
==
```

```
'system'
:
# SystemPrompt ->
return
chat
.
ChatCompletionSystemMessageParam
(
role
=
'system'
,
content
=
message
.
content
)
elif
message
.
role
==
'user'
:
# UserPrompt ->
return
chat
.
ChatCompletionUserMessageParam
(
role
=
'user'
,
content
=
message
.
content
)
elif
message
.
role
==
'tool-return'
:
# ToolReturn ->
return
chat
.
ChatCompletionToolMessageParam
(
role
=
'tool'
,
tool_call_id
=
_guard_tool_call_id
(
t
=
message
,
model_source
=
'OpenAI'
),
content
=
message
.
model_response_str
(),
)
```

```
elif
message
.
role
==
'retry-prompt'
:
# RetryPrompt ->
if
message
.
tool_name
is
None
:
return
chat
.
ChatCompletionUserMessageParam
(
role
=
'user'
,
content
=
message
.
model_response
())
else
:
return
chat
.
ChatCompletionToolMessageParam
(
role
=
'tool'
,
tool_call_id
=
_guard_tool_call_id
(
t
=
message
,
model_source
=
'OpenAI'
),
content
=
message
.
model_response
(),
)
elif
message
.
role
==
'model-text-response'
:
# ModelTextResponse ->
return
chat
.
ChatCompletionAssistantMessageParam
(
role
=
'assistant'
,
```

```
content
=
message
.
content
)
elif
message
.
role
==
'model-structured-response'
:
# ModelStructuredResponse ->
return
chat
.
ChatCompletionAssistantMessageParam
(
role
=
'assistant'
,
tool_calls
=
[
_map_tool_call
(
t
)
for
t
in
message
.
calls
],
)
else
:
assert_never
(
message
)
```

OpenAIStreamTextResponse
dataclass
Bases:
StreamTextResponse
Implementation of
StreamTextResponse
for OpenAI models.
Source code in
pydantic_ai_slim/pydantic_ai/models/openai.py

```
@dataclass
class OpenAIStreamTextResponse(StreamTextResponse):
    """Implementation of `StreamTextResponse` for OpenAI models."""
    _first: str | None
    _response: AsyncStream[ChatCompletionChunk]
    _timestamp: datetime
    _cost: result.Cost
    _buffer: list[str] = field(default_factory=list, init=False)
    async def __anext__(self) -> None:
        if self._first is not None:
            self._buffer
```

```
.
append
(
self
.
_first
)
self
.
_first
=
None
return
None
chunk
=
await
self
.
_response
.
__anext__
()
self
.
_cost
+=
_map_cost
(
chunk
)
try
:
choice
=
chunk
.
choices
[
0
]
except
IndexError
:
raise
StopAsyncIteration
()
# we don't raise StopAsyncIteration on the last chunk because usage comes after this
if
choice
.
finish_reason
is
None
:
assert
choice
.
delta
.
content
is
not
None
,
f
'Expected delta with content, invalid chunk:
{
chunk
!r}
'
if
choice
.
delta
.
content
```

```
is
not
None
:
self
.
_buffer
.
append
(
choice
.
delta
.
content
)
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
Iterable
[
str
]:
yield from
self
.
_buffer
self
.
_buffer
.
clear
()
def
cost
(
self
)
->
Cost
:
return
self
.
_cost
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp
```

OpenAIStreamStructuredResponse
dataclass
Bases:
StreamStructuredResponse
Implementation of
StreamStructuredResponse
for OpenAI models.
Source code in
pydantic_ai_slim/pydantic_ai/models/openai.py

```
@dataclass
class OpenAIStreamStructuredResponse(StreamStructuredResponse):
"""Implementation of `StreamStructuredResponse` for OpenAI models."""
_response: AsyncStream[ChatCompletionChunk]
_delta_tool_calls: dict[int, ChoiceDeltaToolCall]
_timestamp: datetime
_cost: result.Cost
async def
```

```
__anext__
(
self
)
->
None
:
chunk
=
await
self
.
_response
.
__anext__
()
self
.
_cost
+=
_map_cost
(
chunk
)
try
:
choice
=
chunk
.
choices
[
0
]
except
IndexError
:
raise
StopAsyncIteration
()
if
choice
.
finish_reason
is
not
None
:
raise
StopAsyncIteration
()
assert
choice
.
delta
.
content
is
None
,
f
'Expected tool calls, got content instead, invalid chunk:
{
chunk
!r}
'
for
new
in
choice
.
delta
.
tool_calls
or
[]:
if
```

```
current
:=
self
.
_delta_tool_calls
.
get
(
new
.
index
):
if
current
.
function
is
None
:
current
.
function
=
new
.
function
elif
new
.
function
is
not
None
:
current
.
function
.
name
=
_utils
.
add_optional
(
current
.
function
.
name
,
new
.
function
.
name
)
current
.
function
.
arguments
=
_utils
.
add_optional
(
current
.
function
.
arguments
,
new
.
function
.
arguments
```

```
)
else
:
self
.
_delta_tool_calls
[
new
.
index
]
=
new
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
ModelStructuredResponse
:
calls
:
list
[
ToolCall
]
=
[]
for
c
in
self
.
_delta_tool_calls
.
values
():
if
f
:=
c
.
function
:
if
f
.
name
is
not
None
and
f
.
arguments
is
not
None
:
calls
.
append
(
ToolCall
.
from_json
(
f
```

```
.
name
,
f
.
arguments
,
c
.
id
))
return
ModelStructuredResponse
(
calls
,
timestamp
=
self
.
_timestamp
)
def
cost
(
self
)
->
Cost
:
return
self
.
_cost
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp
```


==
Page: Getting Help - PydanticAI
URL: https://ai.pydantic.dev/help/
==

Getting Help - PydanticAI

Debugging and Monitoring
Examples
Examples
Pydantic Model
Weather agent
Bank support
SQL Generation
RAG
Stream markdown
Stream whales
Chat App with FastAPI
API Reference
API Reference
pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Slack
GitHub Issues
Getting Help
If you need help getting started with PydanticAI or with advanced usage, the following sources may be useful.
Slack
Join the
#pydantic-ai
channel in the
Pydantic Slack
to ask questions, get help, and chat about PydanticAI. There's also channels for Pydantic, Logfire, and FastUI.
If you're on a
Logfire
Pro plan, you can also get a dedicated private slack collab channel with us.
GitHub Issues
The
PydanticAI GitHub Issues
are a great place to ask questions and give us feedback.

==
Page: Chat App with FastAPI - PydanticAI
URL: https://ai.pydantic.dev/examples/chat-app/
==

Chat App with FastAPI - PydanticAI
Skip to content
PydanticAI
Chat App with FastAPI
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai
Introduction
Installation & Setup
Getting Help
Contributing
Documentation
Documentation
Agents
Dependencies
Results
Messages and chat history
Testing and Evals
Debugging and Monitoring
Examples
Examples

Pydantic Model
Weather agent
Bank support
SQL Generation
RAG
Stream markdown
Stream whales
Chat App with FastAPI
Chat App with FastAPI
Table of contents
Running the Example
Example Code
API Reference
API Reference
pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Running the Example
Example Code
Introduction
Examples
Chat App with FastAPI
Simple chat app example build with FastAPI.
Demonstrates:
reusing chat history
serializing messages
streaming responses
This demonstrates storing chat history between requests and using it to give the model context for new responses.
Most of the complex logic here is between
chat_app.py
which streams the response to the browser,
and
chat_app.ts
which renders messages in the browser.
Running the Example
With
dependencies installed and environment variables set
, run:
pip
uv
python
-m
pydantic_ai_examples.chat_app
uv
run
-m
pydantic_ai_examples.chat_app
Then open the app at
localhost:8000
.
TODO screenshot.
Example Code
Python code that runs the chat app:
chat_app.py
from
__future__
import
annotations
as
_annotations
import
asyncio
import
sqlite3

```
from
collections.abc
import
AsyncIterator
from
concurrent.futures.thread
import
ThreadPoolExecutor
from
contextlib
import
asynccontextmanager
from
dataclasses
import
dataclass
from
functools
import
partial
from
pathlib
import
Path
from
typing
import
Annotated
,
Any
,
Callable
,
TypeVar
import
fastapi
import
logfire
from
fastapi
import
Depends
,
Request
from
fastapi.responses
import
HTMLResponse
,
Response
,
StreamingResponse
from
pydantic
import
Field
,
TypeAdapter
from
typing_extensions
import
LiteralString
,
ParamSpec
from
pydantic_ai
import
Agent
from
pydantic_ai.messages
import
(
Message
,
MessagesTypeAdapter
,
ModelTextResponse
```

```
,
UserPrompt
,
)
# 'if-token-present' means nothing will be sent (and the example will work) if you don't have
logfire configured
logfire
.
configure
(
send_to_logfire
=
'if-token-present'
)
agent
=
Agent
(
'openai:gpt-4o'
)
THIS_DIR
=
Path
(
__file__
)
.
parent
@asynccontextmanager
async
def
lifespan
(
_app
:
fastapi
.
FastAPI
):
async
with
Database
.
connect
()
as
db
:
yield
{
'db'
:
db
}
app
=
fastapi
.
FastAPI
(
lifespan
=
lifespan
)
logfire
.
instrument_fastapi
(
app
)
@app
.
get
(
'/'
)
async
```

```
def
index
()
->
HTMLResponse
:
return
HTMLResponse
((
THIS_DIR
/
'chat_app.html'
)
.
read_bytes
())
@app
.
get
(
'/chat_app.ts'
)
async
def
main_ts
()
->
Response
:
"""Get the raw typescript code, it's compiled in the browser, forgive me."""
return
Response
((
THIS_DIR
/
'chat_app.ts'
)
.
read_bytes
(),
media_type
=
'text/plain'
)
async
def
get_db
(
request
:
Request
)
->
Database
:
return
request
.
state
.
db
@app
.
get
(
'/chat/'
)
async
def
get_chat
(
database
:
Database
=
Depends
(
```

```
get_db
))
->
Response
:
msgs
=
await
database
.
get_messages
()
return
Response
(
b
'
\n
'
.
join
(
MessageTypeAdapter
.
dump_json
(
m
)
for
m
in
msgs
),
media_type
=
'text/plain'
,
)
@app
.
post
(
'/chat/'
)
async
def
post_chat
(
prompt
:
Annotated
[
str
,
fastapi
.
Form
()],
database
:
Database
=
Depends
(
get_db
)
)
->
StreamingResponse
:
async
def
stream_messages
():
"""Streams new line delimited JSON `Message`s to the client."""
# stream the user prompt so that can be displayed straight away
yield
```

```
MessageTypeAdapter
.
dump_json
(
UserPrompt
(
content
=
prompt
))
+
b
'
\n
'
# get the chat history so far to pass as context to the agent
messages
=
await
database
.
get_messages
()
# run the agent with the user prompt and the chat history
async
with
agent
.
run_stream
(
prompt
,
message_history
=
messages
)
as
result
:
async
for
text
in
result
.
stream
(
debounce_by
=
0.01
):
# text here is a `str` and the frontend wants
# JSON encoded ModelTextResponse, so we create one
m
=
ModelTextResponse
(
content
=
text
,
timestamp
=
result
.
timestamp
())
yield
MessageTypeAdapter
.
dump_json
(
m
)
+
b
'
```

```
\n
'
# add new messages (e.g. the user prompt and the agent response in this case) to the database
await
database
.
add_messages
(
result
.
new_messages_json
())
return
StreamingResponse
(
stream_messages
(),
media_type
=
'text/plain'
)
MessageTypeAdapter
:
TypeAdapter
[
Message
]
=
TypeAdapter
(
Annotated
[
Message
,
Field
(
discriminator
=
'role'
)]
)
P
=
ParamSpec
(
'P'
)
R
=
TypeVar
(
'R'
)
@dataclass
class
Database
:
"""Rudimentary database to store chat messages in SQLite.
The SQLite standard library package is synchronous, so we
use a thread pool executor to run queries asynchronously.
"""
con
:
sqlite3
.
Connection
_loop
:
asyncio
.
AbstractEventLoop
_executor
:
ThreadPoolExecutor
@classmethod
@asynccontextmanager
async
```

```
def
connect
(
cls
,
file
:
Path
=
THIS_DIR
/
'.chat_app_messages.sqlite'
)
->
AsyncIterator
[
Database
]:
with
logfire
.
span
(
'connect to DB'
):
loop
=
asyncio
.
get_event_loop
()
executor
=
ThreadPoolExecutor
(
max_workers
=
1
)
con
=
await
loop
.
run_in_executor
(
executor
,
cls
.
_connect
,
file
)
slf
=
cls
(
con
,
loop
,
executor
)
try
:
yield
slf
finally
:
await
slf
.
_asyncify
(
con
.
```

```
close
)
@staticmethod
def
_connect
(
file
:
Path
)
->
sqlite3
.
Connection
:
con
=
sqlite3
.
connect
(
str
(
file
))
con
=
logfire
.
instrument_sqlite3
(
con
)
cur
=
con
.
cursor
()
cur
.
execute
(
'CREATE TABLE IF NOT EXISTS messages (id INT PRIMARY KEY, message_list TEXT);'
)
con
.
commit
()
return
con
async
def
add_messages
(
self
,
messages
:
bytes
):
await
self
.
_asyncify
(
self
.
_execute
,
'INSERT INTO messages (message_list) VALUES (?);'
,
messages
,
commit
=
True
```

```
,
)
await
self
.
_asyncify
(
self
.
con
.
commit
)
async
def
get_messages
(
self
)
->
list
[
Message
]:
c
=
await
self
.
_asyncify
(
self
.
_execute
,
'SELECT message_list FROM messages order by id desc'
)
rows
=
await
self
.
_asyncify
(
c
.
fetchall
)
messages
:
list
[
Message
]
=
[]
for
row
in
rows
:
messages
.
extend
(
MessagesTypeAdapter
.
validate_json
(
row
[
0
]))
return
messages
def
_execute
```

```
(
self
,
sql
:
LiteralString
,
*
args
:
Any
,
commit
:
bool
=
False
)
->
sqlite3
.
Cursor
:
cur
=
self
.
con
.
cursor
()
cur
.
execute
(
sql
,
args
)
if
commit
:
self
.
con
.
commit
()
return
cur
async
def
_asyncify
(
self
,
func
:
Callable
[
P
,
R
],
*
args
:
P
.
args
,
**
kwargs
:
P
.
kwargs
```

```
)
->
R
:
return
await
self
.
_loop
.
run_in_executor
(
# type: ignore
self
.
_executor
,
partial
(
func
,
**
kwargs
),
*
args
,
# type: ignore
)
if
__name__
==
'__main__'
:
import
uvicorn
uvicorn
.
run
(
'pydantic_ai_examples.chat_app:app'
,
reload
=
True
,
reload_dirs
=
[
str
(
THIS_DIR
)]
)
```

Simple HTML page to render the app:

chat_app.html

```
<!DOCTYPE html>
<
html
lang
=
"en"
>
<
head
>
<
meta
charset
=
"UTF-8"
>
<
meta
name
=
"viewport"
```

```
content
=
"width=device-width, initial-scale=1.0"
>
<
title
>
Chat App
</
title
>
<
link
href
=
"https://cdn.jsdelivr.net/npm/
[email protected]
/dist/css/bootstrap.min.css"
rel
=
"stylesheet"
>
<
style
>
main
{
max-width
:
700
px
;
}
#
conversation
.
user
::
before
{
content
:
'You asked: '
;
font-weight
:
bold
;
display
:
block
;
}
#
conversation
.
llm-response
::
before
{
content
:
'AI Response: '
;
font-weight
:
bold
;
display
:
block
;
}
#
spinner
{
opacity
```

```
:
0
;
transition
:
opacity
500
ms
ease-in
;
width
:
30
px
;
height
:
30
px
;
border
:
3
px
solid
#222
;
border-bottom-color
:
transparent
;
border-radius
:
50
%
;
animation
:
rotation
1
s
linear
infinite
;
}
@
keyframes
rotation
{
0
%
{
transform
:
rotate
(
0
deg
);
}
100
%
{
transform
:
rotate
(
360
deg
);
}
}
#
spinner
.
active
{
```

```
opacity
:
1
;
}
</
style
>
</
head
>
<
body
>
<
main
class
=
"border rounded mx-auto my-5 p-4"
>
<
h1
>
Chat App
</
h1
>
<
p
>
Ask me anything...
</
p
>
<
div
id
=
"conversation"
class
=
"px-2"
></
div
>
<
div
class
=
"d-flex justify-content-center mb-3"
>
<
div
id
=
"spinner"
></
div
>
</
div
>
<
form
method
=
"post"
>
<
input
id
=
"prompt-input"
name
=
"prompt"
class
```

```
=
"form-control"
/>
<
div
class
=
"d-flex justify-content-end"
>
<
button
class
=
"btn btn-primary mt-2"
>
Send
</
button
>
</
div
>
</
form
>
<
div
id
=
"error"
class
=
"d-none text-danger"
>
Error occurred, check the console for more information.
</
div
>
</
main
>
</
body
>
</
html
>
<
script
src
=
"https://cdnjs.cloudflare.com/ajax/libs/typescript/5.6.3/typescript.min.js"
crossorigin
=
"anonymous"
referrerpolicy
=
"no-referrer"
></
script
>
<
script
type
=
"module"
>
// to let me write TypeScript, without adding the burden of npm we do a dirty, non-production-ready
hack
// and transpile the TypeScript code in the browser
// this is (arguably) A neat demo trick, but not suitable for production!
async
function
loadTs
()
{
const
```

```
response
=
await
fetch
(
'/chat_app.ts'
);
const
tsCode
=
await
response
.
text
();
const
jsCode
=
window
.
ts
.
transpile
(
tsCode
,
{
target
:
"es2015"
});
let
script
=
document
.
createElement
(
'script'
);
script
.
type
=
'module'
;
script
.
text
=
jsCode
;
document
.
body
.
appendChild
(
script
);
}
loadTs
().
catch
((
e
)
=>
{
console
.
error
(
e
);
document
.
```

```
getElementById
(
'error'
).
classList
.
remove
(
'd-none'
);
document
.
getElementById
(
'spinner'
).
classList
.
remove
(
'active'
);
});
</
script
>
```

TypeScript to handle rendering the messages, to keep this simple (and at the risk of offending frontend developers) the typescript code is passed to the browser as plain text and transpiled in the browser.

chat_app.ts

```
// BIG FAT WARNING: to avoid the complexity of npm, this typescript is compiled in the browser
// there's currently no static type checking
import
{
marked
}
from
'https://cdnjs.cloudflare.com/ajax/libs/marked/15.0.0/lib/marked.esm.js'
const
convElement
=
document
.
getElementById
(
'conversation'
)
const
promptInput
=
document
.
getElementById
(
'prompt-input'
)
as
HTMLInputElement
const
spinner
=
document
.
getElementById
(
'spinner'
)
// stream the response and render messages as each chunk is received
// data is sent as newline-delimited JSON
async
function
onFetchResponse
(
response
:
Response
)
```

```
:
Promise
<
void
>
{
let
text
=
''
let
decoder
=
new
TextDecoder
()
if
(
response
.
ok
)
{
const
reader
=
response
.
body
.
getReader
()
while
(
true
)
{
const
{
done
,
value
}
=
await
reader
.
read
()
if
(
done
)
{
break
}
text
+=
decoder
.
decode
(
value
)
addMessages
(
text
)
spinner
.
classList
.
remove
(
'active'
)
}
```

```
addMessages
(
text
)
promptInput
.
disabled
=
false
promptInput
.
focus
()
}
else
{
const
text
=
await
response
.
text
()
console
.
error
(
`Unexpected response:
${
response
.
status
}
`
,
{
response
,
text
})
throw
new
Error
(
`Unexpected response:
${
response
.
status
}
`
)
}
}
// The format of messages, this matches pydantic-ai both for brevity and understanding
// in production, you might not want to keep this format all the way to the frontend
interface
Message
{
role
:
string
content
:
string
timestamp
:
string
}
// take raw response text and render messages into the `#conversation` element
// Message timestamp is assumed to be a unique identifier of a message, and is used to deduplicate
// hence you can send data about the same message multiple times, and it will be updated
// instead of creating a new message elements
function
addMessages
(
```

```
responseText
:
string
)
{
const
lines
=
responseText
.
split
(
'\n'
)
const
messages
:
Message
[]
=
lines
.
filter
(
line
=>
line
.
length
>
1
).
map
(
j
=>
JSON
.
parse
(
j
))
for
(
const
message
of
messages
)
{
// we use the timestamp as a crude element id
const
{
timestamp
,
role
,
content
}
=
message
const
id
=
`msg-
${
timestamp
}
`
let
msgDiv
=
document
.
getElementById
(
id
```

```
)
if
(
!
msgDiv
)
{
msgDiv
=
document
.
createElement
(
'div'
)
msgDiv
.
id
=
id
msgDiv
.
title
=
`
${
role
}
at
${
timestamp
}
`
msgDiv
.
classList
.
add
(
'border-top'
,
'pt-2'
,
role
)
convElement
.
appendChild
(
msgDiv
)
}
msgDiv
.
innerHTML
=
marked
.
parse
(
content
)
}
window
.
scrollTo
({
top
:
document.body.scrollHeight
,
behavior
:
'smooth'
})
}
function
```

```
onError
(
error
:
any
)
{
console
.
error
(
error
)
document
.
getElementById
(
'error'
).
classList
.
remove
(
'd-none'
)
document
.
getElementById
(
'spinner'
).
classList
.
remove
(
'active'
)
}
async
function
onSubmit
(
e
:
SubmitEvent
)
:
Promise
<
void
>
{
e
.
preventDefault
()
spinner
.
classList
.
add
(
'active'
)
const
body
=
new
FormData
(
e
.
target
as
HTMLFormElement
)
promptInput
```

```
.
value
=
''
promptInput
.
disabled
=
true
const
response
=
await
fetch
(
'/chat/'
,
{
method
:
'POST'
,
body
})
await
onFetchResponse
(
response
)
}
// call onSubmit when the form is submitted (e.g. user clicks the send button or hits Enter)
document
.
querySelector
(
'form'
).
addEventListener
(
'submit'
,
(
e
)
=>
onSubmit
(
e
).
catch
(
onError
))
// load messages on page load
fetch
(
'/chat/'
).
then
(
onFetchResponse
).
catch
(
(
onError
)
```


==
Page: pydantic_ai.models.anthropic - PydanticAI
URL: https://ai.pydantic.dev/api/models/anthropic/
==

pydantic_ai.models.anthropic - PydanticAI
Skip to content
PydanticAI

pydantic_ai.models.anthropic
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai
Introduction
Installation & Setup
Getting Help
Contributing
Documentation
Documentation
Agents
Dependencies
Results
Messages and chat history
Testing and Evals
Debugging and Monitoring
Examples
Examples
Pydantic Model
Weather agent
Bank support
SQL Generation
RAG
Stream markdown
Stream whales
Chat App with FastAPI
API Reference
API Reference
pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models.anthropic
Table of contents
Setup
anthropic
LatestAnthropicModelNames
AnthropicModelName
AnthropicModel
__init__
AnthropicAgentModel
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Setup
anthropic
LatestAnthropicModelNames
AnthropicModelName
AnthropicModel
__init__
AnthropicAgentModel
Introduction
API Reference
pydantic_ai.models.anthropic
Setup
For details on how to set up authentication with this model, see
model configuration for Anthropic
.
LatestAnthropicModelNames
module-attribute
LatestAnthropicModelNames
=
Literal
[
"claude-3-5-haiku-latest"
,
"claude-3-5-sonnet-latest"
,

"claude-3-opus-latest"
,
]
Latest named Anthropic models.
AnthropicModelName
module-attribute
AnthropicModelName
=
Union
[
str
,
LatestAnthropicModelNames
]
Possible Anthropic model names.
Since Anthropic supports a variety of date-stamped models, we explicitly list the latest models but allow any name in the type hints.
Since
the Anthropic docs
for a full list.
AnthropicModel
dataclass
Bases:
Model
A model that uses the Anthropic API.
Internally, this uses the
Anthropic Python client
to interact with the API.
Apart from
__init__
, all methods are private or match those of the base class.
Note
The
AnthropicModel
class does not yet support streaming responses.
We anticipate adding support for streaming responses in a near-term future release.
Source code in
pydantic_ai_slim/pydantic_ai/models/anthropic.py

```
@dataclass(init=False)
class AnthropicModel(Model):
"""A model that uses the Anthropic API.
Internally, this uses the [Anthropic Python client](https://github.com/anthropics/anthropic-sdk-python) to interact with the API.
Apart from `__init__`, all methods are private or match those of the base class.
!!! note
The `AnthropicModel` class does not yet support streaming responses.
We anticipate adding support for streaming responses in a near-term future release.
"""
model_name: AnthropicModelName
client: AsyncAnthropic = field(repr=False)
def __init__(self, model_name: AnthropicModelName, *
```

```
,
api_key
:
str
|
None
=
None
,
anthropic_client
:
AsyncAnthropic
|
None
=
None
,
http_client
:
AsyncHTTPClient
|
None
=
None
,
):
"""Initialize an Anthropic model.
Args:
model_name: The name of the Anthropic model to use. List of model names available
[here](https://docs.anthropic.com/en/docs/about-claude/models).
api_key: The API key to use for authentication, if not provided, the `ANTHROPIC_API_KEY` environment
variable
will be used if available.
anthropic_client: An existing
[`AsyncAnthropic`](https://github.com/anthropics/anthropic-sdk-python?tab=readme-ov-file#async-
usage)
client to use, if provided, `api_key` and `http_client` must be `None`.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
"""
self
.
model_name
=
model_name
if
anthropic_client
is
not
None
:
assert
http_client
is
None
,
'Cannot provide both `anthropic_client` and `http_client`'
assert
api_key
is
None
,
'Cannot provide both `anthropic_client` and `api_key`'
self
.
client
=
anthropic_client
elif
http_client
is
not
None
:
self
.
client
=
```

```
AsyncAnthropic
(
api_key
=
api_key
,
http_client
=
http_client
)
else
:
self
.
client
=
AsyncAnthropic
(
api_key
=
api_key
,
http_client
=
cached_async_http_client
())
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
],
)
->
AgentModel
:
check_allow_model_requests
()
tools
=
[
self
.
_map_tool_definition
(
r
)
for
r
in
function_tools
]
if
result_tools
:
tools
+=
[
self
```

```
.
_map_tool_definition
(
r
)
for
r
in
result_tools
]
return
AnthropicAgentModel
(
self
.
client
,
self
.
model_name
,
allow_text_result
,
tools
,
)
def
name
(
self
)
->
str
:
return
self
.
model_name
@staticmethod
def
_map_tool_definition
(
f
:
ToolDefinition
)
->
ToolParam
:
return
{
'name'
:
f
.
name
,
'description'
:
f
.
description
,
'input_schema'
:
f
.
parameters_json_schema
,
}
__init__
__init__
(
model_name
:
AnthropicModelName
,
```

*
,
api_key
:
str
|
None
=
None
,
anthropic_client
:
AsyncAnthropic
|
None
=
None
,
http_client
:
AsyncClient
|
None
=
None
)
Initialize an Anthropic model.
Parameters:
Name
Type
Description
Default
model_name
AnthropicModelName
The name of the Anthropic model to use. List of model names available
here
.
required
api_key
str
| None
The API key to use for authentication, if not provided, the
ANTHROPIC_API_KEY
environment variable
will be used if available.
None
anthropic_client
AsyncAnthropic
| None
An existing
AsyncAnthropic
client to use, if provided,
api_key
and
http_client
must be
None
.
None
http_client
AsyncClient
| None
An existing
httpx.AsyncClient
to use for making HTTP requests.
None
Source code in
pydantic_ai_slim/pydantic_ai/models/anthropic.py

```
def __init__(
self,
model_name: AnthropicModelName,
*,
api_key: str | None = None,
anthropic_client: AsyncAnthropic | None = None,
http_client: AsyncHTTPClient | None = None,
):
"""Initialize an Anthropic model.
Args:
model_name: The name of the Anthropic model to use. List of model names available
[here](https://docs.anthropic.com/en/docs/about-claude/models).
api_key: The API key to use for authentication, if not provided, the `ANTHROPIC_API_KEY` environment
variable
will be used if available.
anthropic_client: An existing
[`AsyncAnthropic`](https://github.com/anthropics/anthropic-sdk-python?tab=readme-ov-file#async-
usage)
client to use, if provided, `api_key` and `http_client` must be `None`.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
"""
self.model_name = model_name
if anthropic_client is
```

```
not
None
:
assert
http_client
is
None
,
'Cannot provide both `anthropic_client` and `http_client`'
assert
api_key
is
None
,
'Cannot provide both `anthropic_client` and `api_key`'
self
.
client
=
anthropic_client
elif
http_client
is
not
None
:
self
.
client
=
AsyncAnthropic
(
api_key
=
api_key
,
http_client
=
http_client
)
else
:
self
.
client
=
AsyncAnthropic
(
api_key
=
api_key
,
http_client
=
cached_async_http_client
())
```

AnthropicAgentModel

dataclass

Bases:
AgentModel

Implementation of
AgentModel
for Anthropic models.

Source code in
pydantic_ai_slim/pydantic_ai/models/anthropic.py

```
@dataclass
class
AnthropicAgentModel
(
AgentModel
):
"""Implementation of `AgentModel` for Anthropic models."""
client
:
AsyncAnthropic
model_name
:
str
allow_text_result
:
bool
tools
:
list
[
ToolParam
```

```
]
async
def
request
(
self
,
messages
:
list
[
Message
])
->
tuple
[
ModelAnyResponse
,
result
.
Cost
]:
response
=
await
self
.
_messages_create
(
messages
,
False
)
return
self
.
_process_response
(
response
),
_map_cost
(
response
)
@asynccontextmanager
async
def
request_stream
(
self
,
messages
:
list
[
Message
])
->
AsyncIterator
[
EitherStreamedResponse
]:
response
=
await
self
.
_messages_create
(
messages
,
True
)
async
with
response
:
```

```
yield
await
self
.
_process_streamed_response
(
response
)
@overload
async
def
_messages_create
(
self
,
messages
:
list
[
Message
],
stream
:
Literal
[
True
]
)
->
AsyncStream
[
RawMessageStreamEvent
]:
pass
@overload
async
def
_messages_create
(
self
,
messages
:
list
[
Message
],
stream
:
Literal
[
False
])
->
AnthropicMessage
:
pass
async
def
_messages_create
(
self
,
messages
:
list
[
Message
],
stream
:
bool
)
->
AnthropicMessage
|
AsyncStream
```

```
[
RawMessageStreamEvent
]:
# standalone function to make it easier to override
if
not
self
.
tools
:
tool_choice
:
ToolChoiceParam
|
None
=
None
elif
not
self
.
allow_text_result
:
tool_choice
=
{
'type'
:
'any'
}
else
:
tool_choice
=
{
'type'
:
'auto'
}
system_prompt
:
str
=
''
anthropic_messages
:
list
[
MessageParam
]
=
[]
for
m
in
messages
:
if
m
.
role
==
'system'
:
system_prompt
+=
m
.
content
else
:
anthropic_messages
.
append
(
self
.
```

```
_map_message
(
m
))
return
await
self
.
client
.
messages
.
create
(
max_tokens
=
1024
,
system
=
system_prompt
or
NOT_GIVEN
,
messages
=
anthropic_messages
,
model
=
self
.
model_name
,
temperature
=
0.0
,
tools
=
self
.
tools
or
NOT_GIVEN
,
tool_choice
=
tool_choice
or
NOT_GIVEN
,
stream
=
stream
,
)
@staticmethod
def
_process_response
(
response
:
AnthropicMessage
)
->
ModelAnyResponse
:
"""Process a non-streamed response, and prepare a message to return."""
content
=
response
.
content
if
_all_text_parts
(
```

```
content
):
return
ModelTextResponse
(
content
=
''
.
join
(
b
.
text
for
b
in
content
))
elif
_all_tool_use_parts
(
content
):
return
ModelStructuredResponse
(
[
ToolCall
.
from_dict
(
c
.
name
,
cast
(
dict
[
str
,
Any
],
c
.
input
),
c
.
id
,
)
for
c
in
content
],
)
else
:
# TODO: we plan to support non-homogenous behavior in the future :)
raise
UnexpectedModelBehavior
(
f
'Not yet supported response from Anthropic, expected all parts to be tool calls or text, got
heterogenous:
{
content
!r}
.'
'We anticipate supporting this in a future release.'
)
@staticmethod
async
def
```

```
_process_streamed_response
(
response
:
AsyncStream
[
RawMessageStreamEvent
])
->
EitherStreamedResponse
:
"""TODO: Process a streamed response, and prepare a streaming response to return."""
# We don't yet support streamed responses from Anthropic, so we raise an error here for now.
# Streamed responses will be supported in a future release.
raise
RuntimeError
(
'Streamed responses are not yet supported for Anthropic models.'
)
# Should be returning some sort of AnthropicStreamTextResponse or AnthropicStreamStructuredResponse
# depending on the type of chunk we get, but we need to establish how we handle (and when we get) the
following:
# RawMessageStartEvent
# RawMessageDeltaEvent
# RawMessageStopEvent
# RawContentBlockStartEvent
# RawContentBlockDeltaEvent
# RawContentBlockDeltaEvent
#
# We might refactor streaming internally before we implement this...
@staticmethod
def
_map_message
(
message
:
Message
)
->
MessageParam
:
"""Just maps a `pydantic_ai.Message` to a `anthropic.types.MessageParam`."""
if
message
.
role
==
'user'
:
return
MessageParam
(
role
=
'user'
,
content
=
message
.
content
)
elif
message
.
role
==
'tool-return'
:
return
MessageParam
(
role
=
'user'
,
content
```

```
=
[
ToolResultBlockParam
(
tool_use_id
=
_guard_tool_call_id
(
t
=
message
,
model_source
=
'Anthropic'
),
type
=
'tool_result'
,
content
=
message
.
model_response_str
(),
is_error
=
False
,
)
],
)
elif
message
.
role
==
'retry-prompt'
:
if
message
.
tool_name
is
None
:
return
MessageParam
(
role
=
'user'
,
content
=
message
.
model_response
())
else
:
return
MessageParam
(
role
=
'user'
,
content
=
[
ToolUseBlockParam
(
id
=
_guard_tool_call_id
```

```
(
t
=
message
,
model_source
=
'Anthropic'
),
input
=
message
.
model_response
(),
name
=
message
.
tool_name
,
type
=
'tool_use'
,
),
],
)
elif
message
.
role
==
'model-text-response'
:
return
MessageParam
(
role
=
'assistant'
,
content
=
message
.
content
)
elif
message
.
role
==
'model-structured-response'
:
return
MessageParam
(
role
=
'assistant'
,
content
=
[
_map_tool_call
(
t
)
for
t
in
message
.
calls
])
elif
```

message
.
role
==
'system'
:
raise
UnexpectedModelBehavior
(
'System messages are handled separately for Anthropic, this is a bug, please report it.'
)
else
:
assert_never
(
message
)

==
Page: pydantic_ai.models.ollama - PydanticAI
URL: https://ai.pydantic.dev/api/models/ollama/
==

pydantic_ai.models.ollama - PydanticAI

pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Setup
Example local usage
Example using a remote server
ollama
CommonOllamaModelNames
OllamaModelName
OllamaModel
__init__
Introduction
API Reference
pydantic_ai.models.ollama
Setup
For details on how to set up authentication with this model, see
model configuration for Ollama
.
Example local usage
With
ollama
installed, you can run the server with the model you want to use:
terminal-run-ollama
ollama
run
llama3.2
(this will pull the
llama3.2
model if you don't already have it downloaded)
Then run your code, here's a minimal example:
ollama_example.py

```
from
pydantic
import
BaseModel
from
pydantic_ai
import
Agent
class
CityLocation
(
BaseModel
):
city
:
str
country
:
str
agent
=
Agent
(
'ollama:llama3.2'
,
result_type
=
CityLocation
)
result
=
agent
.
run_sync
(
'Where were the olympics held in 2012?'
)
print
(
result
.
data
)
#> city='London' country='United Kingdom'
```

```
print
(
result
.
cost
())
#> Cost(request_tokens=57, response_tokens=8, total_tokens=65, details=None)
```

Example using a remote server

ollama_example_with_remote_server.py

```
from
pydantic
import
BaseModel
from
pydantic_ai
import
Agent
from
pydantic_ai.models.ollama
import
OllamaModel
ollama_model
=
OllamaModel
(
model_name
=
'qwen2.5-coder:7b'
,
# (1)!
base_url
=
'http://192.168.1.74:11434/v1'
,
# (2)!
)
class
CityLocation
(
BaseModel
):
city
:
str
country
:
str
agent
=
Agent
(
model
=
ollama_model
,
result_type
=
CityLocation
)
result
=
agent
.
run_sync
(
'Where were the olympics held in 2012?'
)
print
(
result
.
data
)
#> city='London' country='United Kingdom'
print
(
result
```

.
cost
())
#> Cost(request_tokens=57, response_tokens=8, total_tokens=65, details=None)
The name of the model running on the remote server
The url of the remote server
See
OllamaModel
for more information
CommonOllamaModelNames
module-attribute
CommonOllamaModelNames
=
Literal
[
"codellama"
,
"gemma"
,
"gemma2"
,
"llama3"
,
"llama3.1"
,
"llama3.2"
,
"llama3.2-vision"
,
"llama3.3"
,
"mistral"
,
"mistral-nemo"
,
"mixtral"
,
"phi3"
,
"qwq"
,
"qwen"
,
"qwen2"
,
"qwen2.5"
,
"starcoder2"
,
]
This contains just the most common ollama models.
For a full list see
ollama.com/library
.
OllamaModelName
module-attribute
OllamaModelName
=
Union
[
CommonOllamaModelNames
,
str
]
Possible ollama models.
Since Ollama supports hundreds of models, we explicitly list the most models but allow any name in the type hints.
OllamaModel
dataclass
Bases:
Model
A model that implements Ollama using the OpenAI API.
Internally, this uses the
OpenAI Python client
to interact with the Ollama server.
Apart from
__init__

, all methods are private or match those of the base class.
Source code in
pydantic_ai_slim/pydantic_ai/models/ollama.py

```
@dataclass
(
init
=
False
)
class
OllamaModel
(
Model
):
"""A model that implements Ollama using the OpenAI API.
Internally, this uses the [OpenAI Python client](https://github.com/openai/openai-python) to
interact with the Ollama server.
```

```
Apart from `__init__`, all methods are private or match those of the base class.
"""
model_name
:
OllamaModelName
openai_model
:
OpenAIModel
def
__init__
(
self
,
model_name
:
OllamaModelName
,
*
,
base_url
:
str
|
None
=
'http://localhost:11434/v1/'
,
openai_client
:
AsyncOpenAI
|
None
=
None
,
http_client
:
AsyncHTTPClient
|
None
=
None
,
):
"""Initialize an Ollama model.
Ollama has built-in compatability for the OpenAI chat completions API ([source]
(https://ollama.com/blog/openai-compatibility)), so we reuse the
[`OpenAIModel`][pydantic_ai.models.openai.OpenAIModel] here.
Args:
model_name: The name of the Ollama model to use. List of models available [here]
(https://ollama.com/library)
You must first download the model (`ollama pull <MODEL-NAME>`) in order to use the model
base_url: The base url for the ollama requests. The default value is the ollama default
openai_client: An existing
[`AsyncOpenAI`](https://github.com/openai/openai-python?tab=readme-ov-file#async-usage)
client to use, if provided, `base_url` and `http_client` must be `None`.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
"""
self
.
model_name
=
model_name
if
openai_client
is
not
None
:
assert
base_url
is
None
,
'Cannot provide both `openai_client` and `base_url`'
assert
http_client
```

```
is
None
,
'Cannot provide both `openai_client` and `http_client`'
self
.
openai_model
=
OpenAIModel
(
model_name
=
model_name
,
openai_client
=
openai_client
)
else
:
# API key is not required for ollama but a value is required to create the client
http_client_
=
http_client
or
cached_async_http_client
()
oai_client
=
AsyncOpenAI
(
base_url
=
base_url
,
api_key
=
'ollama'
,
http_client
=
http_client_
)
self
.
openai_model
=
OpenAIModel
(
model_name
=
model_name
,
openai_client
=
oai_client
)
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
```

```
list
[
ToolDefinition
],
)
->
AgentModel
:
return
await
self
.
openai_model
.
agent_model
(
function_tools
=
function_tools
,
allow_text_result
=
allow_text_result
,
result_tools
=
result_tools
,
)
def
name
(
self
)
->
str
:
return
f
'ollama:
{
self
.
model_name
}
'
__init__
__init__
(
model_name
:
OllamaModelName
,
*
,
base_url
:
str
|
None
=
"http://localhost:11434/v1/"
,
openai_client
:
AsyncOpenAI
|
None
=
None
,
http_client
:
AsyncClient
|
None
=
```

None
)
Initialize an Ollama model.
Ollama has built-in compatability for the OpenAI chat completions API (
source
), so we reuse the
OpenAIModel
here.
Parameters:
Name
Type
Description
Default
model_name
OllamaModelName
The name of the Ollama model to use. List of models available
here
You must first download the model (
ollama pull <MODEL-NAME>
) in order to use the model
required
base_url
str
| None
The base url for the ollama requests. The default value is the ollama default
'http://localhost:11434/v1/'
openai_client
AsyncOpenAI
| None
An existing
AsyncOpenAI
client to use, if provided,
base_url
and
http_client
must be
None
.
None
http_client
AsyncClient
| None
An existing
httpx.AsyncClient
to use for making HTTP requests.
None
Source code in
pydantic_ai_slim/pydantic_ai/models/ollama.py

```
def
__init__
(
self
,
model_name
:
OllamaModelName
,
*
,
base_url
:
str
|
None
=
'http://localhost:11434/v1/'
,
openai_client
:
AsyncOpenAI
|
None
=
None
,
http_client
:
AsyncHTTPClient
|
None
=
None
,
):
"""Initialize an Ollama model.
Ollama has built-in compatability for the OpenAI chat completions API ([source]
(https://ollama.com/blog/openai-compatibility)), so we reuse the
[`OpenAIModel`][pydantic_ai.models.openai.OpenAIModel] here.
Args:
model_name: The name of the Ollama model to use. List of models available [here]
(https://ollama.com/library)
You must first download the model (`ollama pull <MODEL-NAME>`) in order to use the model
base_url: The base url for the ollama requests. The default value is the ollama default
openai_client: An existing
[`AsyncOpenAI`](https://github.com/openai/openai-python?tab=readme-ov-file#async-usage)
client to use, if provided, `base_url` and `http_client` must be `None`.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
"""
self
.
model_name
=
model_name
if
openai_client
is
not
None
:
assert
base_url
is
None
,
'Cannot provide both `openai_client` and `base_url`'
assert
http_client
is
None
,
'Cannot provide both `openai_client` and `http_client`'
self
```

```
.
openai_model
=
OpenAIModel
(
model_name
=
model_name
,
openai_client
=
openai_client
)
else
:
# API key is not required for ollama but a value is required to create the client
http_client_
=
http_client
or
cached_async_http_client
()
oai_client
=
AsyncOpenAI
(
base_url
=
base_url
,
api_key
=
'ollama'
,
http_client
=
http_client_
)
self
.
openai_model
=
OpenAIModel
(
model_name
=
model_name
,
openai_client
=
oai_client
)
```


==
Page: Bank support - PydanticAI
URL: https://ai.pydantic.dev/examples/bank-support/
==

Bank support - PydanticAI
Skip to content
PydanticAI
Bank support
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai
Introduction
Installation & Setup
Getting Help
Contributing
Documentation
Documentation
Agents
Dependencies
Results

Messages and chat history
Testing and Evals
Debugging and Monitoring
Examples
Examples
Pydantic Model
Weather agent
Bank support
Bank support
Table of contents
Running the Example
Example Code
SQL Generation
RAG
Stream markdown
Stream whales
Chat App with FastAPI
API Reference
API Reference
pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Running the Example
Example Code
Introduction
Examples
Bank support
Small but complete example of using PydanticAI to build a support agent for a bank.
Demonstrates:
dynamic system prompt
structured
result_type
tools
Running the Example
With
dependencies installed and environment variables set
, run:
pip
uv
python
-m
pydantic_ai_examples.bank_support
uv
run
-m
pydantic_ai_examples.bank_support
(or
PYDANTIC_AI_MODEL=gemini-1.5-flash ...
)
Example Code
bank_support.py
from
dataclasses
import
dataclass
from
pydantic
import
BaseModel
,
Field
from
pydantic_ai
import
Agent

```
,
RunContext
class
DatabaseConn
:
"""This is a fake database for example purposes.
In reality, you'd be connecting to an external database
(e.g. PostgreSQL) to get information about customers.
"""
@classmethod
async
def
customer_name
(
cls
,
*
,
id
:
int
)
->
str
|
None
:
if
id
==
123
:
return
'John'
@classmethod
async
def
customer_balance
(
cls
,
*
,
id
:
int
,
include_pending
:
bool
)
->
float
:
if
id
==
123
:
return
123.45
else
:
raise
ValueError
(
'Customer not found'
)
@dataclass
class
SupportDependencies
:
customer_id
:
int
db
:
```

```
DatabaseConn
class
SupportResult
(
BaseModel
):
support_advice
:
str
=
Field
(
description
=
'Advice returned to the customer'
)
block_card
:
bool
=
Field
(
description
=
'Whether to block their'
)
risk
:
int
=
Field
(
description
=
'Risk level of query'
,
ge
=
0
,
le
=
10
)
support_agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
SupportDependencies
,
result_type
=
SupportResult
,
system_prompt
=
(
'You are a support agent in our bank, give the '
'customer support and judge the risk level of their query. '
"Reply using the customer's name."
),
)
@support_agent
.
system_prompt
async
def
add_customer_name
(
ctx
:
RunContext
[
```

```
SupportDependencies
])
->
str
:
customer_name
=
await
ctx
.
deps
.
db
.
customer_name
(
id
=
ctx
.
deps
.
customer_id
)
return
f
"The customer's name is
{
customer_name
!r}
"
@support_agent
.
tool
async
def
customer_balance
(
ctx
:
RunContext
[
SupportDependencies
],
include_pending
:
bool
)
->
str
:
"""Returns the customer's current account balance."""
balance
=
await
ctx
.
deps
.
db
.
customer_balance
(
id
=
ctx
.
deps
.
customer_id
,
include_pending
=
include_pending
,
)
return
```

```
f
'$
{
balance
:
.2f
}
'
deps
=
SupportDependencies
(
customer_id
=
123
,
db
=
DatabaseConn
())
result
=
support_agent
.
run_sync
(
'What is my balance?'
,
deps
=
deps
)
print
(
result
.
data
)
"""
support_advice='Hello John, your current account balance, including pending transactions, is
$123.45.' block_card=False risk=1
"""
result
=
support_agent
.
run_sync
(
'I just lost my card!'
,
deps
=
deps
)
print
(
result
.
data
)
"""
support_advice="I'm sorry to hear that, John. We are temporarily blocking your card to prevent
unauthorized transactions." block_card=True risk=8
"""
```


==
Page: Results - PydanticAI
URL: https://ai.pydantic.dev/results/
==

Results - PydanticAI
Skip to content
PydanticAI
Results
Initializing search

pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

Results
Results are the final values returned from
running an agent
.
The result values are wrapped in
RunResult
and
StreamedRunResult
so you can access other data like
cost
of the run and
message history
Both
RunResult
and
StreamedRunResult
are generic in the data they wrap, so typing information about the data returned by the agent is preserved.
olympics.py

```python
from pydantic import BaseModel
from pydantic_ai import Agent

class CityLocation(BaseModel):
    city: str
    country: str

agent = Agent('gemini-1.5-flash', result_type=CityLocation)
result = agent.run_sync('Where were the olympics held in 2012?')
print(result.data)
#> city='London' country='United Kingdom'
print(result.cost())
#> Cost(request_tokens=57, response_tokens=8, total_tokens=65, details=None)
```

(This example is complete, it can be run "as is")

Runs end when either a plain text response is received or the model calls a tool associated with one of the structured result types. We will add limits to make sure a run doesn't go on indefinitely, see #70.

Result data

When the result type is `str`, or a union including `str`, plain text responses are enabled on the model, and the raw text response from the model is used as the response data.

If the result type is a union with multiple members (after remove `str` from the members), each member is registered as a separate tool with the model in order to reduce the complexity of the tool schemas and maximise the changes a model will respond correctly.

If the result type schema is not of type `"object"`, the result type is wrapped in a single element object, so the schema of all tools registered with the model are object schemas.

Structured results (like tools) use Pydantic to build the JSON schema used for the tool, and to validate the data returned by the model.

Bring on PEP-747

Until PEP-747

"Annotating Type Forms" lands, unions are not valid as
type
s in Python.
When creating the agent we need to
type: ignore
the
result_type
argument, and add a type hint to tell type checkers about the type of the agent.
Here's an example of returning either text or a structured value
box_or_error.py

```
from
typing
import
Union
from
pydantic
import
BaseModel
from
pydantic_ai
import
Agent
class
Box
(
BaseModel
):
width
:
int
height
:
int
depth
:
int
units
:
str
agent
:
Agent
[
None
,
Union
[
Box
,
str
]]
=
Agent
(
'openai:gpt-4o-mini'
,
result_type
=
Union
[
Box
,
str
],
# type: ignore
system_prompt
=
(
"Extract me the dimensions of a box, "
"if you can't extract all data, ask the user to try again."
),
)
result
=
agent
.
run_sync
```

```
(
'The box is 10x20x30'
)
print
(
result
.
data
)
#> Please provide the units for the dimensions (e.g., cm, in, m).
result
=
agent
.
run_sync
(
'The box is 10x20x30 cm'
)
print
(
result
.
data
)
#> width=10 height=20 depth=30 units='cm'
```
(This example is complete, it can be run "as is")
Here's an example of using a union return type which registered multiple tools, and wraps non-object schemas in an object:
colors_or_sizes.py
```
from
typing
import
Union
from
pydantic_ai
import
Agent
agent
:
Agent
[
None
,
Union
[
list
[
str
],
list
[
int
]]]
=
Agent
(
'openai:gpt-4o-mini'
,
result_type
=
Union
[
list
[
str
],
list
[
int
]],
# type: ignore
system_prompt
=
'Extract either colors or sizes from the shapes provided.'
,
)
result
```

```
=
agent
.
run_sync
(
'red square, blue circle, green triangle'
)
print
(
result
.
data
)
#> ['red', 'blue', 'green']
result
=
agent
.
run_sync
(
'square size 10, circle size 20, triangle size 30'
)
print
(
result
.
data
)
#> [10, 20, 30]
```

(This example is complete, it can be run "as is")

Result validators functions

Some validation is inconvenient or impossible to do in Pydantic validators, in particular when the validation requires IO and is asynchronous. PydanticAI provides a way to add validation functions via the
agent.result_validator
decorator.

Here's a simplified variant of the
SQL Generation example
:

sql_gen.py

```
from
typing
import
Union
from
fake_database
import
DatabaseConn
,
QueryError
from
pydantic
import
BaseModel
from
pydantic_ai
import
Agent
,
RunContext
,
ModelRetry
class
Success
(
BaseModel
):
sql_query
:
str
class
InvalidRequest
(
BaseModel
):
error_message
:
```

```
str
Response
=
Union
[
Success
,
InvalidRequest
]
agent
:
Agent
[
DatabaseConn
,
Response
]
=
Agent
(
'gemini-1.5-flash'
,
result_type
=
Response
,
# type: ignore
deps_type
=
DatabaseConn
,
system_prompt
=
'Generate PostgreSQL flavored SQL queries based on user input.'
,
)
@agent
.
result_validator
async
def
validate_result
(
ctx
:
RunContext
[
DatabaseConn
],
result
:
Response
)
->
Response
:
if
isinstance
(
result
,
InvalidRequest
):
return
result
try
:
await
ctx
.
deps
.
execute
(
f
'EXPLAIN
{
```

```
result
.
sql_query
}
'
)
except
QueryError
as
e
:
raise
ModelRetry
(
f
'Invalid query:
{
e
}
'
)
from
e
else
:
return
result
result
=
agent
.
run_sync
(
'get me uses who were last active yesterday.'
,
deps
=
DatabaseConn
()
)
print
(
result
.
data
)
#> sql_query='SELECT * FROM users WHERE last_active::date = today() - interval 1 day'
```

(This example is complete, it can be run "as is")

Streamed Results

There two main challenges with streamed results:

Validating structured responses before they're complete, this is achieved by "partial validation" which was recently added to Pydantic in pydantic/pydantic#10748.

When receiving a response, we don't know if it's the final response without starting to stream it and peeking at the content. PydanticAI streams just enough of the response to sniff out if it's a tool call or a result, then streams the whole thing and calls tools, or returns the stream as a StreamedRunResult.

Streaming Text

Example of streamed text result:

streamed_hello_world.py

```
from
pydantic_ai
import
Agent
agent
=
Agent
(
'gemini-1.5-flash'
)
# (1)!
async
def
main
():
```

```
async
with
agent
.
run_stream
(
'Where does "hello world" come from?'
)
as
result
:
# (2)!
async
for
message
in
result
.
stream
():
# (3)!
print
(
message
)
#> The first known
#> The first known use of "hello,
#> The first known use of "hello, world" was in
#> The first known use of "hello, world" was in a 1974 textbook
#> The first known use of "hello, world" was in a 1974 textbook about the C
#> The first known use of "hello, world" was in a 1974 textbook about the C programming language.
```

Streaming works with the standard
Agent
class, and doesn't require any special setup, just a model that supports streaming (currently all models support streaming).
The
Agent.run_stream()
method is used to start a streamed run, this method returns a context manager so the connection can be closed when the stream completes.
Each item yield by
StreamedRunResult.stream()
is the complete text response, extended as new data is received.
(This example is complete, it can be run "as is")
We can also stream text as deltas rather than the entire text in each item:
streamed_delta_hello_world.py

```
from
pydantic_ai
import
Agent
agent
=
Agent
(
'gemini-1.5-flash'
)
async
def
main
():
async
with
agent
.
run_stream
(
'Where does "hello world" come from?'
)
as
result
:
async
for
message
in
result
.
stream_text
```

(
delta
=
True
):
(1)!
print
(
message
)
#> The first known
#> use of "hello,
#> world" was in
#> a 1974 textbook
#> about the C
#> programming language.
stream_text
will error if the response is not text
(This example is complete, it can be run "as is")
Result message not included in
messages
The final result message will
NOT
be added to result messages if you use
.stream_text(delta=True)
,
see
Messages and chat history
for more information.
Streaming Structured Responses
Not all types are supported with partial validation in Pydantic, see
pydantic/pydantic#10748
, generally for model-like structures it's currently best to use
TypeDict
.
Here's an example of streaming a use profile as it's built:
streamed_user_profile.py
from
datetime
import
date
from
typing_extensions
import
TypedDict
from
pydantic_ai
import
Agent
class
UserProfile
(
TypedDict
,
total
=
False
):
name
:
str
dob
:
date
bio
:
str
agent
=
Agent
(
'openai:gpt-4o'
,
result_type
=
UserProfile
,

```
system_prompt
=
'Extract a user profile from the input'
,
)
async
def
main
():
user_input
=
'My name is Ben, I was born on January 28th 1990, I like the chain the dog and the pyramid.'
async
with
agent
.
run_stream
(
user_input
)
as
result
:
async
for
profile
in
result
.
stream
():
print
(
profile
)
#> {'name': 'Ben'}
#> {'name': 'Ben'}
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes'}
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the '}
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyr'}
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyramid'}
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyramid'}
```

(This example is complete, it can be run "as is")

If you want fine-grained control of validation, particularly catching validation errors, you can use the following pattern:

streamed_user_profile.py

```
from
datetime
import
date
from
pydantic
import
ValidationError
from
typing_extensions
import
TypedDict
from
pydantic_ai
import
Agent
class
UserProfile
(
TypedDict
,
total
=
False
):
name
:
str
dob
:
date
```

```
bio
:
str
agent
=
Agent
(
'openai:gpt-4o'
,
result_type
=
UserProfile
)
async
def
main
():
user_input
=
'My name is Ben, I was born on January 28th 1990, I like the chain the dog and the pyramid.'
async
with
agent
.
run_stream
(
user_input
)
as
result
:
async
for
message
,
last
in
result
.
stream_structured
(
debounce_by
=
0.01
):
# (1)!
try
:
profile
=
await
result
.
validate_structured_result
(
# (2)!
message
,
allow_partial
=
not
last
,
)
except
ValidationError
:
continue
print
(
profile
)
#> {'name': 'Ben'}
#> {'name': 'Ben'}
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes'}
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the '}
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyr'}
```

```
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyramid'}
#> {'name': 'Ben', 'dob': date(1990, 1, 28), 'bio': 'Likes the chain the dog and the pyramid'}
```
stream_structured
streams the data as
ModelStructuredResponse
objects, thus iteration can't fail with a
ValidationError
.
validate_structured_result
validates the data,
allow_partial=True
enables pydantic's
experimental_allow_partial
flag on
TypeAdapter
.
(This example is complete, it can be run "as is")
Examples
The following examples demonstrate how to use streamed responses in PydanticAI:
Stream markdown
Stream Whales

==
Page: pydantic_ai.Agent - PydanticAI
URL: https://ai.pydantic.dev/api/agent/
==

pydantic_ai.Agent - PydanticAI

pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents

Introduction
API Reference
pydantic_ai.Agent
Bases:
Generic
[
AgentDeps
,
ResultData
]
Class for defining "agents" - a way to have a specific type of "conversation" with an LLM.
Agents are generic in the dependency type they take
AgentDeps
and the result data type they return,
ResultData
.
By default, if neither generic parameter is customised, agents have type
Agent[None, str]
.
Minimal usage example:

```
from
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
)
result
=
agent
.
run_sync
(
'What is the capital of France?'
)
print
(
result
.
data
)
#> Paris
```

Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```
@final
@dataclass
(
init
=
False
)
```

```
class
Agent
(
Generic
[
AgentDeps
,
ResultData
]):
"""Class for defining "agents" - a way to have a specific type of "conversation" with an LLM.
Agents are generic in the dependency type they take [`AgentDeps`][pydantic_ai.tools.AgentDeps]
and the result data type they return, [`ResultData`][pydantic_ai.result.ResultData].
By default, if neither generic parameter is customised, agents have type `Agent[None, str]`.
Minimal usage example:
```python
from pydantic_ai import Agent
agent = Agent('openai:gpt-4o')
result = agent.run_sync('What is the capital of France?')
print(result.data)
#> Paris
```
"""
we use dataclass fields in order to conveniently know what attributes are available
model
:
models
.
Model
|
models
.
KnownModelName
|
None
"""The default model configured for this agent."""
name
:
str
|
None
"""The name of the agent, used for logging.
If `None`, we try to infer the agent name from the call frame when the agent is first run.
"""
last_run_messages
:
list
[
_messages
.
Message
]
|
None
=
None
"""The messages from the last run, useful when a run raised an exception.
Note: these are not used by the agent, e.g. in future runs, they are just stored for developers'
convenience.
"""
_result_schema
:
_result
.
ResultSchema
[
ResultData
]
|
None
=
field
(
repr
=
False
)
_result_validators
```

```
:
list
[
_result
.
ResultValidator
[
AgentDeps
,
ResultData
]]
=
field
(
repr
=
False
)
_allow_text_result
:
bool
=
field
(
repr
=
False
)
_system_prompts
:
tuple
[
str
,
...
]
=
field
(
repr
=
False
)
_function_tools
:
dict
[
str
,
Tool
[
AgentDeps
]]
=
field
(
repr
=
False
)
_default_retries
:
int
=
field
(
repr
=
False
)
_system_prompt_functions
:
list
[
_system_prompt
.
SystemPromptRunner
```

```
[
AgentDeps
]]
=
field
(
repr
=
False
)
_deps_type
:
type
[
AgentDeps
]
=
field
(
repr
=
False
)
_max_result_retries
:
int
=
field
(
repr
=
False
)
_current_result_retry
:
int
=
field
(
repr
=
False
)
_override_deps
:
_utils
.
Option
[
AgentDeps
]
=
field
(
default
=
None
,
repr
=
False
)
_override_model
:
_utils
.
Option
[
models
.
Model
]
=
field
(
default
=
```

```
None
,
repr
=
False
)
def
__init__
(
self
,
model
:
models
.
Model
|
models
.
KnownModelName
|
None
=
None
,
*
,
result_type
:
type
[
ResultData
]
=
str
,
system_prompt
:
str
|
Sequence
[
str
]
=
(),
deps_type
:
type
[
AgentDeps
]
=
NoneType
,
name
:
str
|
None
=
None
,
retries
:
int
=
1
,
result_tool_name
:
str
=
'final_result'
,
result_tool_description
:
```

```
str
|
None
=
None
,
result_retries
:
int
|
None
=
None
,
tools
:
Sequence
[
Tool
[
AgentDeps
]
|
ToolFuncEither
[
AgentDeps
,
...
]]
=
(),
defer_model_check
:
bool
=
False
,
):
"""Create an agent.
Args:
model: The default model to use for this agent, if not provide,
you must provide the model when calling the agent.
result_type: The type of the result data, used to validate the result data, defaults to `str`.
system_prompt: Static system prompts to use for this agent, you can also register system
prompts via a function with [`system_prompt`][pydantic_ai.Agent.system_prompt].
deps_type: The type used for dependency injection, this parameter exists solely to allow you to
fully
parameterize the agent, and therefore get the best out of static type checking.
If you're not using deps, but want type checking to pass, you can set `deps=None` to satisfy Pyright
or add a type hint `: Agent[None, <return type>]`.
name: The name of the agent, used for logging. If `None`, we try to infer the agent name from the
call frame
when the agent is first run.
retries: The default number of retries to allow before raising an error.
result_tool_name: The name of the tool to use for the final result.
result_tool_description: The description of the final result tool.
result_retries: The maximum number of retries to allow for result validation, defaults to `retries`.
tools: Tools to register with the agent, you can also register tools via the decorators
[`@agent.tool`][pydantic_ai.Agent.tool] and [`@agent.tool_plain`][pydantic_ai.Agent.tool_plain].
defer_model_check: by default, if you provide a [named][pydantic_ai.models.KnownModelName] model,
it's evaluated to create a [`Model`][pydantic_ai.models.Model] instance immediately,
which checks for the necessary environment variables. Set this to `false`
to defer the evaluation until the first run. Useful if you want to
[override the model][pydantic_ai.Agent.override] for testing.
"""
if
model
is
None
or
defer_model_check
:
self
.
model
=
model
```

```
else
:
self
.
model
=
models
.
infer_model
(
model
)
self
.
name
=
name
self
.
_result_schema
=
_result
.
ResultSchema
[
result_type
]
.
build
(
result_type
,
result_tool_name
,
result_tool_description
)
# if the result tool is None, or its schema allows `str`, we allow plain text results
self
.
_allow_text_result
=
self
.
_result_schema
is
None
or
self
.
_result_schema
.
allow_text_result
self
.
_system_prompts
=
(
system_prompt
,)
if
isinstance
(
system_prompt
,
str
)
else
tuple
(
system_prompt
)
self
.
_function_tools
=
{}
self
```

```
.
_default_retries
=
retries
for
tool
in
tools
:
if
isinstance
(
tool
,
Tool
):
self
.
_register_tool
(
tool
)
else
:
self
.
_register_tool
(
Tool
(
tool
))
self
.
_deps_type
=
deps_type
self
.
_system_prompt_functions
=
[]
self
.
_max_result_retries
=
result_retries
if
result_retries
is
not
None
else
retries
self
.
_current_result_retry
=
0
self
.
_result_validators
=
[]
async
def
run
(
self
,
user_prompt
:
str
,
*
,
message_history
```

```
:
list
[
_messages
.
Message
]
|
None
=
None
,
model
:
models
.
Model
|
models
.
KnownModelName
|
None
=
None
,
deps
:
AgentDeps
=
None
,
infer_name
:
bool
=
True
,
)
->
result
.
RunResult
[
ResultData
]:
"""Run the agent with a user prompt in async mode.
Example:
```python
from pydantic_ai import Agent
agent = Agent('openai:gpt-4o')
result_sync = agent.run_sync('What is the capital of Italy?')
print(result_sync.data)
#> Rome
```
Args:
user_prompt: User input to start/continue the conversation.
message_history: History of the conversation so far.
model: Optional model to use for this run, required if `model` was not set when creating the agent.
deps: Optional dependencies to use for this run.
infer_name: Whether to try to infer the agent name from the call frame if it's not set.
Returns:
The result of the run.
"""
if
infer_name
and
self
.
name
is
None
:
self
.
_infer_name
(
```

```
inspect
.
currentframe
())
model_used
,
mode_selection
=
await
self
.
_get_model
(
model
)
deps
=
self
.
_get_deps
(
deps
)
with
_logfire
.
span
(
'
{agent_name}
run {prompt=}'
,
prompt
=
user_prompt
,
agent
=
self
,
mode_selection
=
mode_selection
,
model_name
=
model_used
.
name
(),
agent_name
=
self
.
name
or
'agent'
,
)
as
run_span
:
new_message_index
,
messages
=
await
self
.
_prepare_messages
(
deps
,
user_prompt
,
message_history
)
```

```
self
.
last_run_messages
=
messages
for
tool
in
self
.
_function_tools
.
values
():
tool
.
current_retry
=
0
cost
=
result
.
Cost
()
run_step
=
0
while
True
:
run_step
+=
1
with
_logfire
.
span
(
'preparing model and tools {run_step=}'
,
run_step
=
run_step
):
agent_model
=
await
self
.
_prepare_model
(
model_used
,
deps
)
with
_logfire
.
span
(
'model request'
,
run_step
=
run_step
)
as
model_req_span
:
model_response
,
request_cost
=
await
agent_model
.
```

```
request
(
messages
)
model_req_span
.
set_attribute
(
'response'
,
model_response
)
model_req_span
.
set_attribute
(
'cost'
,
request_cost
)
model_req_span
.
message
=
f
'model request ->
{
model_response
.
role
}
'
messages
.
append
(
model_response
)
cost
+=
request_cost
with
_logfire
.
span
(
'handle model response'
,
run_step
=
run_step
)
as
handle_span
:
final_result
,
response_messages
=
await
self
.
_handle_model_response
(
model_response
,
deps
)
# Add all messages to the conversation
messages
.
extend
(
response_messages
)
# Check if we got a final result
if
```

```
final_result
is
not
None
:
result_data
=
final_result
.
data
run_span
.
set_attribute
(
'all_messages'
,
messages
)
run_span
.
set_attribute
(
'cost'
,
cost
)
handle_span
.
set_attribute
(
'result'
,
result_data
)
handle_span
.
message
=
'handle model response -> final result'
return
result
.
RunResult
(
messages
,
new_message_index
,
result_data
,
cost
)
else
:
# continue the conversation
handle_span
.
set_attribute
(
'tool_responses'
,
response_messages
)
response_msgs
=
' '
.
join
(
r
.
role
for
r
in
response_messages
)
```

```
handle_span
.
message
=
f
'handle model response ->
{
response_msgs
}
'
def
run_sync
(
self
,
user_prompt
:
str
,
*
,
message_history
:
list
[
_messages
.
Message
]
|
None
=
None
,
model
:
models
.
Model
|
models
.
KnownModelName
|
None
=
None
,
deps
:
AgentDeps
=
None
,
infer_name
:
bool
=
True
,
)
->
result
.
RunResult
[
ResultData
]:
"""Run the agent with a user prompt synchronously.
This is a convenience method that wraps `self.run` with `loop.run_until_complete()`.
Example:
```python
from pydantic_ai import Agent
agent = Agent('openai:gpt-4o')
async def main():
result = await agent.run('What is the capital of France?')
print(result.data)
```
```

```
#> Paris
```
```
Args:
user_prompt: User input to start/continue the conversation.
message_history: History of the conversation so far.
model: Optional model to use for this run, required if `model` was not set when creating the agent.
deps: Optional dependencies to use for this run.
infer_name: Whether to try to infer the agent name from the call frame if it's not set.
Returns:
The result of the run.
"""
if
infer_name
and
self
.
name
is
None
:
self
.
_infer_name
(
inspect
.
currentframe
())
loop
=
asyncio
.
get_event_loop
()
return
loop
.
run_until_complete
(
self
.
run
(
user_prompt
,
message_history
=
message_history
,
model
=
model
,
deps
=
deps
,
infer_name
=
False
)
)
@asynccontextmanager
async
def
run_stream
(
self
,
user_prompt
:
str
,
*
,
message_history
:
```

```
list
[
_messages
.
Message
]
|
None
=
None
,
model
:
models
.
Model
|
models
.
KnownModelName
|
None
=
None
,
deps
:
AgentDeps
=
None
,
infer_name
:
bool
=
True
,
)
->
AsyncIterator
[
result
.
StreamedRunResult
[
AgentDeps
,
ResultData
]]:
"""Run the agent with a user prompt in async mode, returning a streamed response.
Example:
```python
from pydantic_ai import Agent
agent = Agent('openai:gpt-4o')
async def main():
async with agent.run_stream('What is the capital of the UK?') as response:
print(await response.get_data())
#> London
```
Args:
user_prompt: User input to start/continue the conversation.
message_history: History of the conversation so far.
model: Optional model to use for this run, required if `model` was not set when creating the agent.
deps: Optional dependencies to use for this run.
infer_name: Whether to try to infer the agent name from the call frame if it's not set.
Returns:
The result of the run.
"""
if
infer_name
and
self
.
name
is
None
:
```

```
# f_back because `asynccontextmanager` adds one frame
if
frame
:=
inspect
.
currentframe
():
# pragma: no branch
self
.
_infer_name
(
frame
.
f_back
)
model_used
,
mode_selection
=
await
self
.
_get_model
(
model
)
deps
=
self
.
_get_deps
(
deps
)
with
_logfire
.
span
(
'
{agent_name}
run stream {prompt=}'
,
prompt
=
user_prompt
,
agent
=
self
,
mode_selection
=
mode_selection
,
model_name
=
model_used
.
name
(),
agent_name
=
self
.
name
or
'agent'
,
)
as
run_span
:
new_message_index
,
```

```
messages
=
await
self
.
_prepare_messages
(
deps
,
user_prompt
,
message_history
)
self
.
last_run_messages
=
messages
for
tool
in
self
.
_function_tools
.
values
():
tool
.
current_retry
=
0
cost
=
result
.
Cost
()
run_step
=
0
while
True
:
run_step
+=
1
with
_logfire
.
span
(
'preparing model and tools {run_step=}'
,
run_step
=
run_step
):
agent_model
=
await
self
.
_prepare_model
(
model_used
,
deps
)
with
_logfire
.
span
(
'model request {run_step=}'
,
run_step
```

```
=
run_step
)
as
model_req_span
:
async
with
agent_model
.
request_stream
(
messages
)
as
model_response
:
model_req_span
.
set_attribute
(
'response_type'
,
model_response
.
__class__
.
__name__
)
# We want to end the "model request" span here, but we can't exit the context manager
# in the traditional way
model_req_span
.
__exit__
(
None
,
None
,
None
)
with
_logfire
.
span
(
'handle model response'
)
as
handle_span
:
final_result
,
response_messages
=
await
self
.
_handle_streamed_model_response
(
model_response
,
deps
)
# Add all messages to the conversation
messages
.
extend
(
response_messages
)
# Check if we got a final result
if
final_result
is
not
None
```

```
:
result_stream
=
final_result
.
data
run_span
.
set_attribute
(
'all_messages'
,
messages
)
handle_span
.
set_attribute
(
'result_type'
,
result_stream
.
__class__
.
__name__
)
handle_span
.
message
=
'handle model response -> final result'
yield
result
.
StreamedRunResult
(
messages
,
new_message_index
,
cost
,
result_stream
,
self
.
_result_schema
,
deps
,
self
.
_result_validators
,
lambda
m
:
run_span
.
set_attribute
(
'all_messages'
,
messages
),
)
return
else
:
# continue the conversation
handle_span
.
set_attribute
(
'tool_responses'
,
response_messages
```

```
)
response_msgs
=
' '
.
join
(
r
.
role
for
r
in
response_messages
)
handle_span
.
message
=
f
'handle model response ->
{
response_msgs
}
'
# the model_response should have been fully streamed by now, we can add it's cost
cost
+=
model_response
.
cost
()
@contextmanager
def
override
(
self
,
*
,
deps
:
AgentDeps
|
_utils
.
Unset
=
_utils
.
UNSET
,
model
:
models
.
Model
|
models
.
KnownModelName
|
_utils
.
Unset
=
_utils
.
UNSET
,
)
->
Iterator
[
None
]:
"""Context manager to temporarily override agent dependencies and model.
```

```
This is particularly useful when testing.
You can find an example of this [here](../testing-evals.md#overriding-model-via-pytest-fixtures).
Args:
deps: The dependencies to use instead of the dependencies passed to the agent run.
model: The model to use instead of the model passed to the agent run.
"""
if
_utils
.
is_set
(
deps
):
override_deps_before
=
self
.
_override_deps
self
.
_override_deps
=
_utils
.
Some
(
deps
)
else
:
override_deps_before
=
_utils
.
UNSET
# noinspection PyTypeChecker
if
_utils
.
is_set
(
model
):
override_model_before
=
self
.
_override_model
# noinspection PyTypeChecker
self
.
_override_model
=
_utils
.
Some
(
models
.
infer_model
(
model
))
# pyright: ignore[reportArgumentType]
else
:
override_model_before
=
_utils
.
UNSET
try
:
yield
finally
:
if
```

```
_utils
.
is_set
(
override_deps_before
):
self
.
_override_deps
=
override_deps_before
if
_utils
.
is_set
(
override_model_before
):
self
.
_override_model
=
override_model_before
@overload
def
system_prompt
(
self
,
func
:
Callable
[[
RunContext
[
AgentDeps
]],
str
],
/
)
->
Callable
[[
RunContext
[
AgentDeps
]],
str
]:
...
@overload
def
system_prompt
(
self
,
func
:
Callable
[[
RunContext
[
AgentDeps
]],
Awaitable
[
str
]],
/
)
->
Callable
[[
RunContext
[
AgentDeps
```

```
]],
Awaitable
[
str
]]:
...
@overload
def
system_prompt
(
self
,
func
:
Callable
[[],
str
],
/
)
->
Callable
[[],
str
]:
...
@overload
def
system_prompt
(
self
,
func
:
Callable
[[],
Awaitable
[
str
]],
/
)
->
Callable
[[],
Awaitable
[
str
]]:
...
def
system_prompt
(
self
,
func
:
_system_prompt
.
SystemPromptFunc
[
AgentDeps
],
/
)
->
_system_prompt
.
SystemPromptFunc
[
AgentDeps
]:
"""Decorator to register a system prompt function.
Optionally takes [`RunContext`][pydantic_ai.tools.RunContext] as its only argument.
Can decorate a sync or async functions.
Overloads for every possible signature of `system_prompt` are included so the decorator doesn't
obscure
```

```
the type of the function, see `tests/typed_agent.py` for tests.
Example:
```python
from pydantic_ai import Agent, RunContext
agent = Agent('test', deps_type=str)
@agent.system_prompt
def simple_system_prompt() -> str:
return 'foobar'
@agent.system_prompt
async def async_system_prompt(ctx: RunContext[str]) -> str:
return f'{ctx.deps} is the best'
result = agent.run_sync('foobar', deps='spam')
print(result.data)
#> success (no tool calls)
```
"""
self
.
_system_prompt_functions
.
append
(
_system_prompt
.
SystemPromptRunner
(
func
))
return
func
@overload
def
result_validator
(
self
,
func
:
Callable
[[
RunContext
[
AgentDeps
],
ResultData
],
ResultData
],
/
)
->
Callable
[[
RunContext
[
AgentDeps
],
ResultData
],
ResultData
]:
...
@overload
def
result_validator
(
self
,
func
:
Callable
[[
RunContext
[
AgentDeps
],
ResultData
```

```
],
Awaitable
[
ResultData
]],
/
)
->
Callable
[[
RunContext
[
AgentDeps
],
ResultData
],
Awaitable
[
ResultData
]]:
...
@overload
def
result_validator
(
self
,
func
:
Callable
[[
ResultData
],
ResultData
],
/
)
->
Callable
[[
ResultData
],
ResultData
]:
...
@overload
def
result_validator
(
self
,
func
:
Callable
[[
ResultData
],
Awaitable
[
ResultData
]],
/
)
->
Callable
[[
ResultData
],
Awaitable
[
ResultData
]]:
...
def
result_validator
(
self
```

```
,
func
:
_result
.
ResultValidatorFunc
[
AgentDeps
,
ResultData
],
/
)
->
_result
.
ResultValidatorFunc
[
AgentDeps
,
ResultData
]:
"""Decorator to register a result validator function.
Optionally takes [`RunContext`][pydantic_ai.tools.RunContext] as its first argument.
Can decorate a sync or async functions.
Overloads for every possible signature of `result_validator` are included so the decorator doesn't
obscure
the type of the function, see `tests/typed_agent.py` for tests.
Example:
```python
from pydantic_ai import Agent, ModelRetry, RunContext
agent = Agent('test', deps_type=str)
@agent.result_validator
def result_validator_simple(data: str) -> str:
if 'wrong' in data:
raise ModelRetry('wrong response')
return data
@agent.result_validator
async def result_validator_deps(ctx: RunContext[str], data: str) -> str:
if ctx.deps in data:
raise ModelRetry('wrong response')
return data
result = agent.run_sync('foobar', deps='spam')
print(result.data)
#> success (no tool calls)
```
"""
self
.
_result_validators
.
append
(
_result
.
ResultValidator
(
func
))
return
func
@overload
def
tool
(
self
,
func
:
ToolFuncContext
[
AgentDeps
,
ToolParams
],
/
)
```

```
->
ToolFuncContext
[
AgentDeps
,
ToolParams
]:
...
@overload
def
tool
(
self
,
/
,
*
,
retries
:
int
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
,
)
->
Callable
[[
ToolFuncContext
[
AgentDeps
,
ToolParams
]],
ToolFuncContext
[
AgentDeps
,
ToolParams
]]:
...
def
tool
(
self
,
func
:
ToolFuncContext
[
AgentDeps
,
ToolParams
]
|
None
=
None
,
/
,
*
,
retries
```

```
:
int
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
,
)
->
Any
:
"""Decorator to register a tool function which takes [`RunContext`][pydantic_ai.tools.RunContext] as
its first argument.
Can decorate a sync or async functions.
The docstring is inspected to extract both the tool description and description of each parameter,
[learn more](../agents.md#function-tools-and-schema).
We can't add overloads for every possible signature of tool, since the return type is a recursive
union
so the signature of functions decorated with `@agent.tool` is obscured.
Example:
```python
from pydantic_ai import Agent, RunContext
agent = Agent('test', deps_type=int)
@agent.tool
def foobar(ctx: RunContext[int], x: int) -> int:
return ctx.deps + x
@agent.tool(retries=2)
async def spam(ctx: RunContext[str], y: float) -> float:
return ctx.deps + y
result = agent.run_sync('foobar', deps=1)
print(result.data)
#> {"foobar":1,"spam":1.0}
```
Args:
func: The tool function to register.
retries: The number of retries to allow for this tool, defaults to the agent's default retries,
which defaults to 1.
prepare: custom method to prepare the tool definition for each step, return `None` to omit this
tool from a given step. This is useful if you want to customise a tool at call time,
or omit it completely from a step. See [`ToolPrepareFunc`][pydantic_ai.tools.ToolPrepareFunc].
"""
if
func
is
None
:
def
tool_decorator
(
func_
:
ToolFuncContext
[
AgentDeps
,
ToolParams
],
)
->
ToolFuncContext
[
AgentDeps
,
ToolParams
]:
noinspection PyTypeChecker
```

```
self
.
_register_function
(
func_
,
True
,
retries
,
prepare
)
return
func_
return
tool_decorator
else
:
# noinspection PyTypeChecker
self
.
_register_function
(
func
,
True
,
retries
,
prepare
)
return
func
@overload
def
tool_plain
(
self
,
func
:
ToolFuncPlain
[
ToolParams
],
/
)
->
ToolFuncPlain
[
ToolParams
]:
...
@overload
def
tool_plain
(
self
,
/
,
*
,
retries
:
int
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
```

```
|
None
=
None
,
)
->
Callable
[[
ToolFuncPlain
[
ToolParams
]],
ToolFuncPlain
[
ToolParams
]]:
...
def
tool_plain
(
self
,
func
:
ToolFuncPlain
[
ToolParams
]
|
None
=
None
,
/
,
*
,
retries
:
int
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
,
)
->
Any
:
"""Decorator to register a tool function which DOES NOT take `RunContext` as an argument.
Can decorate a sync or async functions.
The docstring is inspected to extract both the tool description and description of each parameter,
[learn more](../agents.md#function-tools-and-schema).
We can't add overloads for every possible signature of tool, since the return type is a recursive
union
so the signature of functions decorated with `@agent.tool` is obscured.
Example:
```python
from pydantic_ai import Agent, RunContext
agent = Agent('test')
@agent.tool
def foobar(ctx: RunContext[int]) -> int:
return 123
@agent.tool(retries=2)
async def spam(ctx: RunContext[str]) -> float:
```
```

```
return 3.14
result = agent.run_sync('foobar', deps=1)
print(result.data)
#> {"foobar":123,"spam":3.14}
```
```
Args:
func: The tool function to register.
retries: The number of retries to allow for this tool, defaults to the agent's default retries,
which defaults to 1.
prepare: custom method to prepare the tool definition for each step, return `None` to omit this
tool from a given step. This is useful if you want to customise a tool at call time,
or omit it completely from a step. See [`ToolPrepareFunc`][pydantic_ai.tools.ToolPrepareFunc].
"""
if
func
is
None
:
def
tool_decorator
(
func_
:
ToolFuncPlain
[
ToolParams
])
->
ToolFuncPlain
[
ToolParams
]:
# noinspection PyTypeChecker
self
.
_register_function
(
func_
,
False
,
retries
,
prepare
)
return
func_
return
tool_decorator
else
:
self
.
_register_function
(
func
,
False
,
retries
,
prepare
)
return
func
def
_register_function
(
self
,
func
:
ToolFuncEither
[
AgentDeps
,
ToolParams
```

```
],
takes_ctx
:
bool
,
retries
:
int
|
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
,
)
->
None
:
"""Private utility to register a function as a tool."""
retries_
=
retries
if
retries
is
not
None
else
self
.
_default_retries
tool
=
Tool
(
func
,
takes_ctx
=
takes_ctx
,
max_retries
=
retries_
,
prepare
=
prepare
)
self
.
_register_tool
(
tool
)
def
_register_tool
(
self
,
tool
:
Tool
[
AgentDeps
])
->
None
:
"""Private utility to register a tool instance."""
if
```

```
tool
.
max_retries
is
None
:
# noinspection PyTypeChecker
tool
=
dataclasses
.
replace
(
tool
,
max_retries
=
self
.
_default_retries
)
if
tool
.
name
in
self
.
_function_tools
:
raise
exceptions
.
UserError
(
f
'Tool name conflicts with existing tool:
{
tool
.
name
!r}
'
)
if
self
.
_result_schema
and
tool
.
name
in
self
.
_result_schema
.
tools
:
raise
exceptions
.
UserError
(
f
'Tool name conflicts with result schema name:
{
tool
.
name
!r}
'
)
self
.
_function_tools
[
```

```
tool
.
name
]
=
tool
async
def
_get_model
(
self
,
model
:
models
.
Model
|
models
.
KnownModelName
|
None
)
->
tuple
[
models
.
Model
,
str
]:
"""Create a model configured for this agent.
Args:
model: model to use for this run, required if `model` was not set when creating the agent.
Returns:
a tuple of `(model used, how the model was selected)`
"""
model_
:
models
.
Model
if
some_model
:=
self
.
_override_model
:
# we don't want `override()` to cover up errors from the model not being defined, hence this check
if
model
is
None
and
self
.
model
is
None
:
raise
exceptions
.
UserError
(
'`model` must be set either when creating the agent or when calling it. '
'(Even when `override(model=...)` is customizing the model that will actually be called)'
)
model_
=
some_model
.
value
mode_selection
```

```
=
'override-model'
elif
model
is
not
None
:
model_
=
models
.
infer_model
(
model
)
mode_selection
=
'custom'
elif
self
.
model
is
not
None
:
# noinspection PyTypeChecker
model_
=
self
.
model
=
models
.
infer_model
(
self
.
model
)
mode_selection
=
'from-agent'
else
:
raise
exceptions
.
UserError
(
'`model` must be set either when creating the agent or when calling it.'
)
return
model_
,
mode_selection
async
def
_prepare_model
(
self
,
model
:
models
.
Model
,
deps
:
AgentDeps
)
->
models
.
```

```
AgentModel
:
"""Create building tools and create an agent model."""
function_tools
:
list
[
ToolDefinition
]
=
[]
async
def
add_tool
(
tool
:
Tool
[
AgentDeps
])
->
None
:
ctx
=
RunContext
(
deps
,
tool
.
current_retry
,
tool
.
name
)
if
tool_def
:=
await
tool
.
prepare_tool_def
(
ctx
):
function_tools
.
append
(
tool_def
)
await
asyncio
.
gather
(
*
map
(
add_tool
,
self
.
_function_tools
.
values
()))
return
await
model
.
agent_model
(
function_tools
```

```
=
function_tools
,
allow_text_result
=
self
.
_allow_text_result
,
result_tools
=
self
.
_result_schema
.
tool_defs
()
if
self
.
_result_schema
is
not
None
else
[],
)
async
def
_prepare_messages
(
self
,
deps
:
AgentDeps
,
user_prompt
:
str
,
message_history
:
list
[
_messages
.
Message
]
|
None
)
->
tuple
[
int
,
list
[
_messages
.
Message
]]:
# if message history includes system prompts, we don't want to regenerate them
if
message_history
and
any
(
m
.
role
==
'system'
for
m
in
```

```
message_history
):
# shallow copy messages
messages
=
message_history
.
copy
()
else
:
messages
=
await
self
.
_init_messages
(
deps
)
if
message_history
:
messages
+=
message_history
new_message_index
=
len
(
messages
)
messages
.
append
(
_messages
.
UserPrompt
(
user_prompt
))
return
new_message_index
,
messages
async
def
_handle_model_response
(
self
,
model_response
:
_messages
.
ModelAnyResponse
,
deps
:
AgentDeps
)
->
tuple
[
_MarkFinalResult
[
ResultData
]
|
None
,
list
[
_messages
.
Message
```

```
]]:
"""Process a non-streamed response from the model.
Returns:
A tuple of `(final_result, messages)`. If `final_result` is not `None`, the conversation should end.
"""
if
model_response
.
role
==
'model-text-response'
:
# plain string response
if
self
.
_allow_text_result
:
result_data_input
=
cast
(
ResultData
,
model_response
.
content
)
try
:
result_data
=
await
self
.
_validate_result
(
result_data_input
,
deps
,
None
)
except
_result
.
ToolRetryError
as
e
:
self
.
_incr_result_retry
()
return
None
,
[
e
.
tool_retry
]
else
:
return
_MarkFinalResult
(
result_data
),
[]
else
:
self
.
_incr_result_retry
()
response
```

```
=
_messages
.
RetryPrompt
(
content
=
'Plain text responses are not permitted, please call one of the functions instead.'
,
)
return
None
,
[
response
]
elif
model_response
.
role
==
'model-structured-response'
:
if
self
.
_result_schema
is
not
None
:
# if there's a result schema, and any of the calls match one of its tools, return the result
# NOTE: this means we ignore any other tools called here
if
match
:=
self
.
_result_schema
.
find_tool
(
model_response
):
call
,
result_tool
=
match
try
:
result_data
=
result_tool
.
validate
(
call
)
result_data
=
await
self
.
_validate_result
(
result_data
,
deps
,
call
)
except
_result
.
ToolRetryError
as
```

```
e
:
self
.
_incr_result_retry
()
return
None
,
[
e
.
tool_retry
]
else
:
# Add a ToolReturn message for the schema tool call
tool_return
=
_messages
.
ToolReturn
(
tool_name
=
call
.
tool_name
,
content
=
'Final result processed.'
,
tool_call_id
=
call
.
tool_call_id
,
)
return
_MarkFinalResult
(
result_data
),
[
tool_return
]
if
not
model_response
.
calls
:
raise
exceptions
.
UnexpectedModelBehavior
(
'Received empty tool call message'
)
# otherwise we run all tool functions in parallel
messages
:
list
[
_messages
.
Message
]
=
[]
tasks
:
list
[
asyncio
```

```
.
Task
[
_messages
.
Message
]]
=
[]
for
call
in
model_response
.
calls
:
if
tool
:=
self
.
_function_tools
.
get
(
call
.
tool_name
):
tasks
.
append
(
asyncio
.
create_task
(
tool
.
run
(
deps
,
call
),
name
=
call
.
tool_name
))
else
:
messages
.
append
(
self
.
_unknown_tool
(
call
.
tool_name
))
with
_logfire
.
span
(
'running {tools=}'
,
tools
=
[
t
.
```

```
get_name
()
for
t
in
tasks
]):
task_results
:
Sequence
[
_messages
.
Message
]
=
await
asyncio
.
gather
(
*
tasks
)
messages
.
extend
(
task_results
)
return
None
,
messages
else
:
assert_never
(
model_response
)
async
def
_handle_streamed_model_response
(
self
,
model_response
:
models
.
EitherStreamedResponse
,
deps
:
AgentDeps
)
->
tuple
[
_MarkFinalResult
[
models
.
EitherStreamedResponse
]
|
None
,
list
[
_messages
.
Message
]]:
"""Process a streamed response from the model.
Returns:
A tuple of (final_result, messages). If final_result is not None, the conversation should end.
```

```
"""
if
isinstance
(
model_response
,
models
.
StreamTextResponse
):
# plain string response
if
self
.
_allow_text_result
:
return
_MarkFinalResult
(
model_response
),
[]
else
:
self
.
_incr_result_retry
()
response
=
_messages
.
RetryPrompt
(
content
=
'Plain text responses are not permitted, please call one of the functions instead.'
,
)
# stream the response, so cost is correct
async
for
_
in
model_response
:
pass
return
None
,
[
response
]
else
:
assert
isinstance
(
model_response
,
models
.
StreamStructuredResponse
),
f
'Unexpected response:
{
model_response
}
'
if
self
.
_result_schema
is
not
None
```

```
:
# if there's a result schema, iterate over the stream until we find at least one tool
# NOTE: this means we ignore any other tools called here
structured_msg
=
model_response
.
get
()
while
not
structured_msg
.
calls
:
try
:
await
model_response
.
__anext__
()
except
StopAsyncIteration
:
break
structured_msg
=
model_response
.
get
()
if
match
:=
self
.
_result_schema
.
find_tool
(
structured_msg
):
call
,
_
=
match
tool_return
=
_messages
.
ToolReturn
(
tool_name
=
call
.
tool_name
,
content
=
'Final result processed.'
,
tool_call_id
=
call
.
tool_call_id
,
)
return
_MarkFinalResult
(
model_response
),
[
```

```
tool_return
]
# the model is calling a tool function, consume the response to get the next message
async
for
_
in
model_response
:
pass
structured_msg
=
model_response
.
get
()
if
not
structured_msg
.
calls
:
raise
exceptions
.
UnexpectedModelBehavior
(
'Received empty tool call message'
)
messages
:
list
[
_messages
.
Message
]
=
[
structured_msg
]
# we now run all tool functions in parallel
tasks
:
list
[
asyncio
.
Task
[
_messages
.
Message
]]
=
[]
for
call
in
structured_msg
.
calls
:
if
tool
:=
self
.
_function_tools
.
get
(
call
.
tool_name
):
tasks
```

```
.
append
(
asyncio
.
create_task
(
tool
.
run
(
deps
,
call
),
name
=
call
.
tool_name
))
else
:
messages
.
append
(
self
.
_unknown_tool
(
call
.
tool_name
))
with
_logfire
.
span
(
'running {tools=}'
,
tools
=
[
t
.
get_name
()
for
t
in
tasks
]):
task_results
:
Sequence
[
_messages
.
Message
]
=
await
asyncio
.
gather
(
*
tasks
)
messages
.
extend
(
task_results
)
```

```
return
None
,
messages
async
def
_validate_result
(
self
,
result_data
:
ResultData
,
deps
:
AgentDeps
,
tool_call
:
_messages
.
ToolCall
|
None
)
->
ResultData
:
for
validator
in
self
.
_result_validators
:
result_data
=
await
validator
.
validate
(
result_data
,
deps
,
self
.
_current_result_retry
,
tool_call
)
return
result_data
def
_incr_result_retry
(
self
)
->
None
:
self
.
_current_result_retry
+=
1
if
self
.
_current_result_retry
>
self
.
_max_result_retries
:
```

```
raise
exceptions
.
UnexpectedModelBehavior
(
f
'Exceeded maximum retries (
{
self
.
_max_result_retries
}
) for result validation'
)
async
def
_init_messages
(
self
,
deps
:
AgentDeps
)
->
list
[
_messages
.
Message
]:
"""Build the initial messages for the conversation."""
messages
:
list
[
_messages
.
Message
]
=
[
_messages
.
SystemPrompt
(
p
)
for
p
in
self
.
_system_prompts
]
for
sys_prompt_runner
in
self
.
_system_prompt_functions
:
prompt
=
await
sys_prompt_runner
.
run
(
deps
)
messages
.
append
(
_messages
.
```

```
SystemPrompt
(
prompt
))
return
messages
def
_unknown_tool
(
self
,
tool_name
:
str
)
->
_messages
.
RetryPrompt
:
self
.
_incr_result_retry
()
names
=
list
(
self
.
_function_tools
.
keys
())
if
self
.
_result_schema
:
names
.
extend
(
self
.
_result_schema
.
tool_names
())
if
names
:
msg
=
f
'Available tools:
{
", "
.
join
(
names
)
}
'
else
:
msg
=
'No tools available.'
return
_messages
.
RetryPrompt
(
content
=
```

```
f
'Unknown tool name:
{
tool_name
!r}
.
{
msg
}
'
)
def
_get_deps
(
self
,
deps
:
AgentDeps
)
->
AgentDeps
:
"""Get deps for a run.
If we've overridden deps via `_override_deps`, use that, otherwise use the deps passed to the call.
We could do runtime type checking of deps against `self._deps_type`, but that's a slippery slope.
"""
if
some_deps
:=
self
.
_override_deps
:
return
some_deps
.
value
else
:
return
deps
def
_infer_name
(
self
,
function_frame
:
FrameType
|
None
)
->
None
:
"""Infer the agent name from the call frame.
Usage should be `self._infer_name(inspect.currentframe())`.
"""
assert
self
.
name
is
None
,
'Name already set'
if
function_frame
is
not
None
:
# pragma: no branch
if
parent_frame
:=
```

```
function_frame
.
f_back
:
# pragma: no branch
for
name
,
item
in
parent_frame
.
f_locals
.
items
():
if
item
is
self
:
self
.
name
=
name
return
if
parent_frame
.
f_locals
!=
parent_frame
.
f_globals
:
# if we couldn't find the agent in locals and globals are a different dict, try globals
for
name
,
item
in
parent_frame
.
f_globals
.
items
():
if
item
is
self
:
self
.
name
=
name
return
__init__
__init__
(
model
:
Model
|
KnownModelName
|
None
=
None
,
*
,
result_type
:
type
```

```
[
ResultData
]
=
str
,
system_prompt
:
str
|
Sequence
[
str
]
=
(),
deps_type
:
type
[
AgentDeps
]
=
NoneType
,
name
:
str
|
None
=
None
,
retries
:
int
=
1
,
result_tool_name
:
str
=
"final_result"
,
result_tool_description
:
str
|
None
=
None
,
result_retries
:
int
|
None
=
None
,
tools
:
Sequence
[
Tool
[
AgentDeps
]
|
ToolFuncEither
[
AgentDeps
,
...
]
]
```

=
(),
defer_model_check
:
bool
=
False
)
Create an agent.
Parameters:
Name
Type
Description
Default
model
Model
|
KnownModelName
| None
The default model to use for this agent, if not provide,
you must provide the model when calling the agent.
None
result_type
type
[
ResultData
]
The type of the result data, used to validate the result data, defaults to
str
.
str
system_prompt
str
|
Sequence
[
str
]
Static system prompts to use for this agent, you can also register system
prompts via a function with
system_prompt
.
()
deps_type
type
[
AgentDeps
]
The type used for dependency injection, this parameter exists solely to allow you to fully
parameterize the agent, and therefore get the best out of static type checking.
If you're not using deps, but want type checking to pass, you can set
deps=None
to satisfy Pyright
or add a type hint
: Agent[None, <return type>]
.
NoneType
name
str
| None
The name of the agent, used for logging. If
None
, we try to infer the agent name from the call frame
when the agent is first run.
None
retries
int
The default number of retries to allow before raising an error.
1
result_tool_name
str
The name of the tool to use for the final result.
'final_result'
result_tool_description
str
| None
The description of the final result tool.

None
result_retries
int
| None
The maximum number of retries to allow for result validation, defaults to
retries
.
None
tools
Sequence
[
Tool
[
AgentDeps
] |
ToolFuncEither
[
AgentDeps
, ...]]
Tools to register with the agent, you can also register tools via the decorators
@agent.tool
and
@agent.tool_plain
.
()
defer_model_check
bool
by default, if you provide a
named
model,
it's evaluated to create a
Model
instance immediately,
which checks for the necessary environment variables. Set this to
false
to defer the evaluation until the first run. Useful if you want to
override the model
for testing.
False
Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```
def __init__(
    self,
    model: models.Model | models.KnownModelName | None = None,
    *,
    result_type: type[ResultData] = str,
    system_prompt: str | Sequence[str] = (),
    deps_type: type[AgentDeps] =
```

```
NoneType
,
name
:
str
|
None
=
None
,
retries
:
int
=
1
,
result_tool_name
:
str
=
'final_result'
,
result_tool_description
:
str
|
None
=
None
,
result_retries
:
int
|
None
=
None
,
tools
:
Sequence
[
Tool
[
AgentDeps
]
|
ToolFuncEither
[
AgentDeps
,
...
]]
=
(),
defer_model_check
:
bool
=
False
,
):
"""Create an agent.
Args:
model: The default model to use for this agent, if not provide,
you must provide the model when calling the agent.
result_type: The type of the result data, used to validate the result data, defaults to `str`.
system_prompt: Static system prompts to use for this agent, you can also register system
prompts via a function with [`system_prompt`][pydantic_ai.Agent.system_prompt].
deps_type: The type used for dependency injection, this parameter exists solely to allow you to
fully
parameterize the agent, and therefore get the best out of static type checking.
If you're not using deps, but want type checking to pass, you can set `deps=None` to satisfy Pyright
or add a type hint `: Agent[None, <return type>]`.
name: The name of the agent, used for logging. If `None`, we try to infer the agent name from the
call frame
when the agent is first run.
```

```
retries: The default number of retries to allow before raising an error.
result_tool_name: The name of the tool to use for the final result.
result_tool_description: The description of the final result tool.
result_retries: The maximum number of retries to allow for result validation, defaults to `retries`.
tools: Tools to register with the agent, you can also register tools via the decorators
[`@agent.tool`][pydantic_ai.Agent.tool] and [`@agent.tool_plain`][pydantic_ai.Agent.tool_plain].
defer_model_check: by default, if you provide a [named][pydantic_ai.models.KnownModelName] model,
it's evaluated to create a [`Model`][pydantic_ai.models.Model] instance immediately,
which checks for the necessary environment variables. Set this to `false`
to defer the evaluation until the first run. Useful if you want to
[override the model][pydantic_ai.Agent.override] for testing.
"""
if
model
is
None
or
defer_model_check
:
self
.
model
=
model
else
:
self
.
model
=
models
.
infer_model
(
model
)
self
.
name
=
name
self
.
_result_schema
=
_result
.
ResultSchema
[
result_type
]
.
build
(
result_type
,
result_tool_name
,
result_tool_description
)
# if the result tool is None, or its schema allows `str`, we allow plain text results
self
.
_allow_text_result
=
self
.
_result_schema
is
None
or
self
.
_result_schema
.
allow_text_result
self
```

```
.
_system_prompts
=
(
system_prompt
,)
if
isinstance
(
system_prompt
,
str
)
else
tuple
(
system_prompt
)
self
.
_function_tools
=
{}
self
.
_default_retries
=
retries
for
tool
in
tools
:
if
isinstance
(
tool
,
Tool
):
self
.
_register_tool
(
tool
)
else
:
self
.
_register_tool
(
Tool
(
tool
))
self
.
_deps_type
=
deps_type
self
.
_system_prompt_functions
=
[]
self
.
_max_result_retries
=
result_retries
if
result_retries
is
not
None
else
```

retries
self
.
_current_result_retry
=
0
self
.
_result_validators
=
[]
name
instance-attribute
name
:
str
|
None
=
name
The name of the agent, used for logging.
If
None
, we try to infer the agent name from the call frame when the agent is first run.
run
async
run
(
user_prompt
:
str
,
*
,
message_history
:
list
[
Message
]
|
None
=
None
,
model
:
Model
|
KnownModelName
|
None
=
None
,
deps
:
AgentDeps
=
None
,
infer_name
:
bool
=
True
)
->
RunResult
[
ResultData
]
Run the agent with a user prompt in async mode.
Example:
from
pydantic_ai
import

Agent
agent
=
Agent
(
'openai:gpt-4o'
)
result_sync
=
agent
.
run_sync
(
'What is the capital of Italy?'
)
print
(
result_sync
.
data
)
#> Rome
Parameters:
Name
Type
Description
Default
user_prompt
str
User input to start/continue the conversation.
required
message_history
list
[
Message
] | None
History of the conversation so far.
None
model
Model
|
KnownModelName
| None
Optional model to use for this run, required if
model
was not set when creating the agent.
None
deps
AgentDeps
Optional dependencies to use for this run.
None
infer_name
bool
Whether to try to infer the agent name from the call frame if it's not set.
True
Returns:
Type
Description
RunResult
[
ResultData
]
The result of the run.
Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```
async
```

```
def
run
(
self
,
user_prompt
:
str
,
*
,
message_history
:
list
[
_messages
.
Message
]
|
None
=
None
,
model
:
models
.
Model
|
models
.
KnownModelName
|
None
=
None
,
deps
:
AgentDeps
=
None
,
infer_name
:
bool
=
True
,
)
->
result
.
RunResult
[
ResultData
]:
"""Run the agent with a user prompt in async mode.
Example:
```python
from pydantic_ai import Agent
agent = Agent('openai:gpt-4o')
result_sync = agent.run_sync('What is the capital of Italy?')
print(result_sync.data)
#> Rome
```
Args:
user_prompt: User input to start/continue the conversation.
message_history: History of the conversation so far.
model: Optional model to use for this run, required if `model` was not set when creating the agent.
deps: Optional dependencies to use for this run.
infer_name: Whether to try to infer the agent name from the call frame if it's not set.
Returns:
The result of the run.
"""
if
```

```
infer_name
and
self
.
name
is
None
:
self
.
_infer_name
(
inspect
.
currentframe
())
model_used
,
mode_selection
=
await
self
.
_get_model
(
model
)
deps
=
self
.
_get_deps
(
deps
)
with
_logfire
.
span
(
'
{agent_name}
run {prompt=}'
,
prompt
=
user_prompt
,
agent
=
self
,
mode_selection
=
mode_selection
,
model_name
=
model_used
.
name
(),
agent_name
=
self
.
name
or
'agent'
,
)
as
run_span
:
new_message_index
,
messages
```

```
=
await
self
.
_prepare_messages
(
deps
,
user_prompt
,
message_history
)
self
.
last_run_messages
=
messages
for
tool
in
self
.
_function_tools
.
values
():
tool
.
current_retry
=
0
cost
=
result
.
Cost
()
run_step
=
0
while
True
:
run_step
+=
1
with
_logfire
.
span
(
'preparing model and tools {run_step=}'
,
run_step
=
run_step
):
agent_model
=
await
self
.
_prepare_model
(
model_used
,
deps
)
with
_logfire
.
span
(
'model request'
,
run_step
=
```

```
run_step
)
as
model_req_span
:
model_response
,
request_cost
=
await
agent_model
.
request
(
messages
)
model_req_span
.
set_attribute
(
'response'
,
model_response
)
model_req_span
.
set_attribute
(
'cost'
,
request_cost
)
model_req_span
.
message
=
f
'model request ->
{
model_response
.
role
}
'
messages
.
append
(
model_response
)
cost
+=
request_cost
with
_logfire
.
span
(
'handle model response'
,
run_step
=
run_step
)
as
handle_span
:
final_result
,
response_messages
=
await
self
.
_handle_model_response
(
model_response
```

```
,
deps
)
# Add all messages to the conversation
messages
.
extend
(
response_messages
)
# Check if we got a final result
if
final_result
is
not
None
:
result_data
=
final_result
.
data
run_span
.
set_attribute
(
'all_messages'
,
messages
)
run_span
.
set_attribute
(
'cost'
,
cost
)
handle_span
.
set_attribute
(
'result'
,
result_data
)
handle_span
.
message
=
'handle model response -> final result'
return
result
.
RunResult
(
messages
,
new_message_index
,
result_data
,
cost
)
else
:
# continue the conversation
handle_span
.
set_attribute
(
'tool_responses'
,
response_messages
)
response_msgs
=
```

' '
.
join
(
r
.
role
for
r
in
response_messages
)
handle_span
.
message
=
f
'handle model response ->
{
response_msgs
}
'
run_sync
run_sync
(
user_prompt
:
str
,
*
,
message_history
:
list
[
Message
]
|
None
=
None
,
model
:
Model
|
KnownModelName
|
None
=
None
,
deps
:
AgentDeps
=
None
,
infer_name
:
bool
=
True
)
->
RunResult
[
ResultData
]
Run the agent with a user prompt synchronously.
This is a convenience method that wraps
self.run
with
loop.run_until_complete()
.
Example:
from

```
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
)
async
def
main
():
result
=
await
agent
.
run
(
'What is the capital of France?'
)
print
(
result
.
data
)
#> Paris
```

Parameters:

Name	Type	Description	Default
user_prompt	str	User input to start/continue the conversation.	required
message_history	list [Message] | None	History of the conversation so far.	None
model	Model | KnownModelName | None	Optional model to use for this run, required if model was not set when creating the agent.	None
deps	AgentDeps	Optional dependencies to use for this run.	None
infer_name	bool	Whether to try to infer the agent name from the call frame if it's not set.	True

Returns:

Type	Description
RunResult [ResultData]	The result of the run.

Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```
def run_sync(self, user_prompt: str, *, message_history: list[_messages.Message] | None = None, model: models.Model | models.KnownModelName | None = None, deps: AgentDeps
```

```
=
None
,
infer_name
:
bool
=
True
,
)
->
result
.
RunResult
[
ResultData
]:
"""Run the agent with a user prompt synchronously.
This is a convenience method that wraps `self.run` with `loop.run_until_complete()`.
Example:
```python
from pydantic_ai import Agent
agent = Agent('openai:gpt-4o')
async def main():
result = await agent.run('What is the capital of France?')
print(result.data)
#> Paris
```
Args:
user_prompt: User input to start/continue the conversation.
message_history: History of the conversation so far.
model: Optional model to use for this run, required if `model` was not set when creating the agent.
deps: Optional dependencies to use for this run.
infer_name: Whether to try to infer the agent name from the call frame if it's not set.
Returns:
The result of the run.
"""
if
infer_name
and
self
.
name
is
None
:
self
.
_infer_name
(
inspect
.
currentframe
())
loop
=
asyncio
.
get_event_loop
()
return
loop
.
run_until_complete
(
self
.
run
(
user_prompt
,
message_history
=
message_history
,
model
=
```

```
model
,
deps
=
deps
,
infer_name
=
False
)
)
run_stream
async
run_stream
(
user_prompt
:
str
,
*
,
message_history
:
list
[
Message
]
|
None
=
None
,
model
:
Model
|
KnownModelName
|
None
=
None
,
deps
:
AgentDeps
=
None
,
infer_name
:
bool
=
True
)
->
AsyncIterator
[
StreamedRunResult
[
AgentDeps
,
ResultData
]
]
Run the agent with a user prompt in async mode, returning a streamed response.
Example:
from
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
)
async
```

```
def
main
():
async
with
agent
.
run_stream
(
'What is the capital of the UK?'
)
as
response
:
print
(
await
response
.
get_data
())
#> London
```

Parameters:
Name
Type
Description
Default
user_prompt
str
User input to start/continue the conversation.
required
message_history
list
[
Message
] | None
History of the conversation so far.
None
model
Model
|
KnownModelName
| None
Optional model to use for this run, required if
model
was not set when creating the agent.
None
deps
AgentDeps
Optional dependencies to use for this run.
None
infer_name
bool
Whether to try to infer the agent name from the call frame if it's not set.
True
Returns:
Type
Description
AsyncIterator
[
StreamedRunResult
[
AgentDeps
,
ResultData
]]
The result of the run.
Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```
@asynccontextmanager
async def run_stream(
    self,
    user_prompt: str,
    *,
    message_history: list[_messages.Message] | None = None,
    model: models.Model | models.KnownModelName | None = None,
    deps: AgentDeps = None,
    infer_name: bool = True,
) -> AsyncIterator[result.
```

```
StreamedRunResult
[
AgentDeps
,
ResultData
]]:
"""Run the agent with a user prompt in async mode, returning a streamed response.
Example:
```python
from pydantic_ai import Agent
agent = Agent('openai:gpt-4o')
async def main():
async with agent.run_stream('What is the capital of the UK?') as response:
print(await response.get_data())
#> London
```
Args:
user_prompt: User input to start/continue the conversation.
message_history: History of the conversation so far.
model: Optional model to use for this run, required if `model` was not set when creating the agent.
deps: Optional dependencies to use for this run.
infer_name: Whether to try to infer the agent name from the call frame if it's not set.
Returns:
The result of the run.
"""
if
infer_name
and
self
.
name
is
None
:
f_back because `asynccontextmanager` adds one frame
if
frame
:=
inspect
.
currentframe
():
pragma: no branch
self
.
_infer_name
(
frame
.
f_back
)
model_used
,
mode_selection
=
await
self
.
_get_model
(
model
)
deps
=
self
.
_get_deps
(
deps
)
with
_logfire
.
span
(
'
{agent_name}
```

```
run stream {prompt=}'
,
prompt
=
user_prompt
,
agent
=
self
,
mode_selection
=
mode_selection
,
model_name
=
model_used
.
name
(),
agent_name
=
self
.
name
or
'agent'
,
)
as
run_span
:
new_message_index
,
messages
=
await
self
.
_prepare_messages
(
deps
,
user_prompt
,
message_history
)
self
.
last_run_messages
=
messages
for
tool
in
self
.
_function_tools
.
values
():
tool
.
current_retry
=
0
cost
=
result
.
Cost
()
run_step
=
0
while
True
```

```
:
run_step
+=
1
with
_logfire
.
span
(
'preparing model and tools {run_step=}'
,
run_step
=
run_step
):
agent_model
=
await
self
.
_prepare_model
(
model_used
,
deps
)
with
_logfire
.
span
(
'model request {run_step=}'
,
run_step
=
run_step
)
as
model_req_span
:
async
with
agent_model
.
request_stream
(
messages
)
as
model_response
:
model_req_span
.
set_attribute
(
'response_type'
,
model_response
.
__class__
.
__name__
)
# We want to end the "model request" span here, but we can't exit the context manager
# in the traditional way
model_req_span
.
__exit__
(
None
,
None
,
None
)
with
_logfire
```

```
.
span
(
'handle model response'
)
as
handle_span
:
final_result
,
response_messages
=
await
self
.
_handle_streamed_model_response
(
model_response
,
deps
)
# Add all messages to the conversation
messages
.
extend
(
response_messages
)
# Check if we got a final result
if
final_result
is
not
None
:
result_stream
=
final_result
.
data
run_span
.
set_attribute
(
'all_messages'
,
messages
)
handle_span
.
set_attribute
(
'result_type'
,
result_stream
.
__class__
.
__name__
)
handle_span
.
message
=
'handle model response -> final result'
yield
result
.
StreamedRunResult
(
messages
,
new_message_index
,
cost
,
result_stream
```

```
,
self
.
_result_schema
,
deps
,
self
.
_result_validators
,
lambda
m
:
run_span
.
set_attribute
(
'all_messages'
,
messages
),
)
return
else
:
# continue the conversation
handle_span
.
set_attribute
(
'tool_responses'
,
response_messages
)
response_msgs
=
' '
.
join
(
r
.
role
for
r
in
response_messages
)
handle_span
.
message
=
f
'handle model response ->
{
response_msgs
}
'
# the model_response should have been fully streamed by now, we can add it's cost
cost
+=
model_response
.
cost
()
```

model
instance-attribute
model
:
Model
|
KnownModelName
|
None
The default model configured for this agent.
override

override
(
*
,
deps
:
AgentDeps
|
Unset
=
UNSET
,
model
:
Model
|
KnownModelName
|
Unset
=
UNSET
)
->
Iterator
[
None
]
Context manager to temporarily override agent dependencies and model.
This is particularly useful when testing.
You can find an example of this
here
.
Parameters:

Name	Type	Description	Default
deps	AgentDeps | Unset	The dependencies to use instead of the dependencies passed to the agent run.	UNSET
model	Model | KnownModelName | Unset	The model to use instead of the model passed to the agent run.	UNSET

Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```python
@contextmanager
def override(
    self,
    *,
    deps: AgentDeps | _utils.Unset = _utils.UNSET,
    model: models.Model | models.KnownModelName | _utils.Unset = _utils.UNSET,
) -> Iterator[None]:
    """Context manager to temporarily override agent dependencies and model.
    This is particularly useful when testing.
    You can find an example of this [here](../testing-evals.md#overriding-model-via-pytest-fixtures).
    Args:
    deps: The dependencies to use instead of the dependencies passed to the agent run.
    model: The model to use instead of the model passed to the agent run.
    """
    if _utils.is_set(deps):
        override_deps_before = self._override_deps
        self
```

```
.
_override_deps
=
_utils
.
Some
(
deps
)
else
:
override_deps_before
=
_utils
.
UNSET
# noinspection PyTypeChecker
if
_utils
.
is_set
(
model
):
override_model_before
=
self
.
_override_model
# noinspection PyTypeChecker
self
.
_override_model
=
_utils
.
Some
(
models
.
infer_model
(
model
))
# pyright: ignore[reportArgumentType]
else
:
override_model_before
=
_utils
.
UNSET
try
:
yield
finally
:
if
_utils
.
is_set
(
override_deps_before
):
self
.
_override_deps
=
override_deps_before
if
_utils
.
is_set
(
override_model_before
):
self
```

.
_override_model
=
override_model_before
last_run_messages
class-attribute
instance-attribute
last_run_messages
:
list
[
Message
]
|
None
=
None
The messages from the last run, useful when a run raised an exception.
Note: these are not used by the agent, e.g. in future runs, they are just stored for developers' convenience.
system_prompt
system_prompt
(
func
:
Callable
[[
RunContext
[
AgentDeps
]],
str
]
)
->
Callable
[[
RunContext
[
AgentDeps
]],
str
]
system_prompt
(
func
:
Callable
[[
RunContext
[
AgentDeps
]],
Awaitable
[
str
]]
)
->
Callable
[[
RunContext
[
AgentDeps
]],
Awaitable
[
str
]]
system_prompt
(
func
:
Callable
[[],
str
])

```
->
Callable
[[],
str
]
system_prompt
(
func
:
Callable
[[],
Awaitable
[
str
]]
)
->
Callable
[[],
Awaitable
[
str
]]
system_prompt
(
func
:
SystemPromptFunc
[
AgentDeps
],
)
->
SystemPromptFunc
[
AgentDeps
]
```

Decorator to register a system prompt function.
Optionally takes
RunContext
as its only argument.
Can decorate a sync or async functions.
Overloads for every possible signature of
system_prompt
are included so the decorator doesn't obscure
the type of the function, see
tests/typed_agent.py
for tests.
Example:

```
from
pydantic_ai
import
Agent
,
RunContext
agent
=
Agent
(
'test'
,
deps_type
=
str
)
@agent
.
system_prompt
def
simple_system_prompt
()
->
str
:
return
'foobar'
@agent
```

```
.
system_prompt
async
def
async_system_prompt
(
ctx
:
RunContext
[
str
])
->
str
:
return
f
'
{
ctx
.
deps
}
is the best'
result
=
agent
.
run_sync
(
'foobar'
,
deps
=
'spam'
)
print
(
result
.
data
)
#> success (no tool calls)
```

Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```
def
system_prompt
(
self
,
func
:
_system_prompt
.
SystemPromptFunc
[
AgentDeps
],
/
)
->
_system_prompt
.
SystemPromptFunc
[
AgentDeps
]:
"""Decorator to register a system prompt function.
Optionally takes [`RunContext`][pydantic_ai.tools.RunContext] as its only argument.
Can decorate a sync or async functions.
Overloads for every possible signature of `system_prompt` are included so the decorator doesn't
obscure
the type of the function, see `tests/typed_agent.py` for tests.
Example:
```python
from pydantic_ai import Agent, RunContext
agent = Agent('test', deps_type=str)
@agent.system_prompt
def simple_system_prompt() -> str:
return 'foobar'
@agent.system_prompt
async def async_system_prompt(ctx: RunContext[str]) -> str:
return f'{ctx.deps} is the best'
result = agent.run_sync('foobar', deps='spam')
print(result.data)
#> success (no tool calls)
```
"""
self
.
_system_prompt_functions
.
append
(
_system_prompt
.
SystemPromptRunner
(
func
))
return
func
tool
tool
(
func
:
ToolFuncContext
[
AgentDeps
,
ToolParams
]
)
->
ToolFuncContext
[
AgentDeps
,
ToolParams
]
tool
```

```
(
*
,
retries
:
int
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
)
->
Callable
[
[
ToolFuncContext
[
AgentDeps
,
ToolParams
]],
ToolFuncContext
[
AgentDeps
,
ToolParams
],
]
tool
(
func
:
(
ToolFuncContext
[
AgentDeps
,
ToolParams
]
|
None
)
=
None
,
/
,
*
,
retries
:
int
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
```

None
,
)
->
Any
Decorator to register a tool function which takes
RunContext
as its first argument.
Can decorate a sync or async functions.
The docstring is inspected to extract both the tool description and description of each parameter,
learn more
.
We can't add overloads for every possible signature of tool, since the return type is a recursive
union
so the signature of functions decorated with
@agent.tool
is obscured.
Example:
```
from
pydantic_ai
import
Agent
,
RunContext
agent
=
Agent
(
'test'
,
deps_type
=
int
)
@agent
.
tool
def
foobar
(
ctx
:
RunContext
[
int
],
x
:
int
)
->
int
:
return
ctx
.
deps
+
x
@agent
.
tool
(
retries
=
2
)
async
def
spam
(
ctx
:
RunContext
[
str
],
```

y
:
float
)
->
float
:
return
ctx
.
deps
+
y
result
=
agent
.
run_sync
(
'foobar'
,
deps
=
1
)
print
(
result
.
data
)
#> {"foobar":1,"spam":1.0}
Parameters:
Name
Type
Description
Default
func
ToolFuncContext
[
AgentDeps
,
ToolParams
] | None
The tool function to register.
None
retries
int
| None
The number of retries to allow for this tool, defaults to the agent's default retries,
which defaults to 1.
None
prepare
ToolPrepareFunc
[
AgentDeps
] | None
custom method to prepare the tool definition for each step, return
None
to omit this
tool from a given step. This is useful if you want to customise a tool at call time,
or omit it completely from a step. See
ToolPrepareFunc
.
None
Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```
def tool(self, func: ToolFuncContext[AgentDeps, ToolParams] | None = None, /, *, retries: int | None =
```

```
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
,
)
->
Any
:
"""Decorator to register a tool function which takes [`RunContext`][pydantic_ai.tools.RunContext] as
its first argument.
Can decorate a sync or async functions.
The docstring is inspected to extract both the tool description and description of each parameter,
[learn more](../agents.md#function-tools-and-schema).
We can't add overloads for every possible signature of tool, since the return type is a recursive
union
so the signature of functions decorated with `@agent.tool` is obscured.
Example:
```python
from pydantic_ai import Agent, RunContext
agent = Agent('test', deps_type=int)
@agent.tool
def foobar(ctx: RunContext[int], x: int) -> int:
return ctx.deps + x
@agent.tool(retries=2)
async def spam(ctx: RunContext[str], y: float) -> float:
return ctx.deps + y
result = agent.run_sync('foobar', deps=1)
print(result.data)
#> {"foobar":1,"spam":1.0}
```
Args:
func: The tool function to register.
retries: The number of retries to allow for this tool, defaults to the agent's default retries,
which defaults to 1.
prepare: custom method to prepare the tool definition for each step, return `None` to omit this
tool from a given step. This is useful if you want to customise a tool at call time,
or omit it completely from a step. See [`ToolPrepareFunc`][pydantic_ai.tools.ToolPrepareFunc].
"""
if
func
is
None
:
def
tool_decorator
(
func_
:
ToolFuncContext
[
AgentDeps
,
ToolParams
],
)
->
ToolFuncContext
[
AgentDeps
,
ToolParams
]:
noinspection PyTypeChecker
self
.
_register_function
(
func_
```

```
,
True
,
retries
,
prepare
)
return
func_
return
tool_decorator
else
:
# noinspection PyTypeChecker
self
.
_register_function
(
func
,
True
,
retries
,
prepare
)
return
func
tool_plain
tool_plain
(
func
:
ToolFuncPlain
[
ToolParams
],
)
->
ToolFuncPlain
[
ToolParams
]
tool_plain
(
*
,
retries
:
int
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
)
->
Callable
[
[
ToolFuncPlain
[
ToolParams
]],
ToolFuncPlain
[
ToolParams
```

```
]
]
tool_plain
(
func
:
ToolFuncPlain
[
ToolParams
]
|
None
=
None
,
/
,
*
,
retries
:
int
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
,
)
->
Any
```

Decorator to register a tool function which DOES NOT take
RunContext
as an argument.
Can decorate a sync or async functions.
The docstring is inspected to extract both the tool description and description of each parameter,
learn more
.
We can't add overloads for every possible signature of tool, since the return type is a recursive
union
so the signature of functions decorated with
@agent.tool
is obscured.
Example:

```
from
pydantic_ai
import
Agent
,
RunContext
agent
=
Agent
(
'test'
)
@agent
.
tool
def
foobar
(
ctx
:
RunContext
[
int
```

])
->
int
:
return
123
@agent
.
tool
(
retries
=
2
)
async
def
spam
(
ctx
:
RunContext
[
str
])
->
float
:
return
3.14
result
=
agent
.
run_sync
(
'foobar'
,
deps
=
1
)
print
(
result
.
data
)
#> {"foobar":123,"spam":3.14}
Parameters:
Name
Type
Description
Default
func
ToolFuncPlain
[
ToolParams
] | None
The tool function to register.
None
retries
int
| None
The number of retries to allow for this tool, defaults to the agent's default retries,
which defaults to 1.
None
prepare
ToolPrepareFunc
[
AgentDeps
] | None
custom method to prepare the tool definition for each step, return
None
to omit this
tool from a given step. This is useful if you want to customise a tool at call time,
or omit it completely from a step. See
ToolPrepareFunc

.
None
Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```
def tool_plain(self, func: ToolFuncPlain[ToolParams] | None = None, /
```

```
,
*
,
retries
:
int
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
,
)
->
Any
:
"""Decorator to register a tool function which DOES NOT take `RunContext` as an argument.
Can decorate a sync or async functions.
The docstring is inspected to extract both the tool description and description of each parameter,
[learn more](../agents.md#function-tools-and-schema).
We can't add overloads for every possible signature of tool, since the return type is a recursive
union
so the signature of functions decorated with `@agent.tool` is obscured.
Example:
```python
from pydantic_ai import Agent, RunContext
agent = Agent('test')
@agent.tool
def foobar(ctx: RunContext[int]) -> int:
return 123
@agent.tool(retries=2)
async def spam(ctx: RunContext[str]) -> float:
return 3.14
result = agent.run_sync('foobar', deps=1)
print(result.data)
#> {"foobar":123,"spam":3.14}
```
Args:
func: The tool function to register.
retries: The number of retries to allow for this tool, defaults to the agent's default retries,
which defaults to 1.
prepare: custom method to prepare the tool definition for each step, return `None` to omit this
tool from a given step. This is useful if you want to customise a tool at call time,
or omit it completely from a step. See [`ToolPrepareFunc`][pydantic_ai.tools.ToolPrepareFunc].
"""
if
func
is
None
:
def
tool_decorator
(
func_
:
ToolFuncPlain
[
ToolParams
])
->
ToolFuncPlain
[
ToolParams
]:
noinspection PyTypeChecker
self
.
```

```
_register_function
(
func_
,
False
,
retries
,
prepare
)
return
func_
return
tool_decorator
else
:
self
.
_register_function
(
func
,
False
,
retries
,
prepare
)
return
func
result_validator
result_validator
(
func
:
Callable
[
[
RunContext
[
AgentDeps
],
ResultData
],
ResultData
]
)
->
Callable
[
[
RunContext
[
AgentDeps
],
ResultData
],
ResultData
]
result_validator
(
func
:
Callable
[
[
RunContext
[
AgentDeps
],
ResultData
],
Awaitable
[
ResultData
],
]
```

)
->
Callable
[
[
RunContext
[
AgentDeps
],
ResultData
],
Awaitable
[
ResultData
],
]
result_validator
(
func
:
Callable
[[
ResultData
],
ResultData
]
)
->
Callable
[[
ResultData
],
ResultData
]
result_validator
(
func
:
Callable
[[
ResultData
],
Awaitable
[
ResultData
]]
)
->
Callable
[[
ResultData
],
Awaitable
[
ResultData
]]
result_validator
(
func
:
ResultValidatorFunc
[
AgentDeps
,
ResultData
]
)
->
ResultValidatorFunc
[
AgentDeps
,
ResultData
]
Decorator to register a result validator function.
Optionally takes
RunContext

as its first argument.
Can decorate a sync or async functions.
Overloads for every possible signature of
result_validator
are included so the decorator doesn't obscure
the type of the function, see
tests/typed_agent.py
for tests.
Example:

```
from
pydantic_ai
import
Agent
,
ModelRetry
,
RunContext
agent
=
Agent
(
'test'
,
deps_type
=
str
)
@agent
.
result_validator
def
result_validator_simple
(
data
:
str
)
->
str
:
if
'wrong'
in
data
:
raise
ModelRetry
(
'wrong response'
)
return
data
@agent
.
result_validator
async
def
result_validator_deps
(
ctx
:
RunContext
[
str
],
data
:
str
)
->
str
:
if
ctx
.
deps
in
```

```
data
:
raise
ModelRetry
(
'wrong response'
)
return
data
result
=
agent
.
run_sync
(
'foobar'
,
deps
=
'spam'
)
print
(
result
.
data
)
#> success (no tool calls)
```

Source code in
pydantic_ai_slim/pydantic_ai/agent.py

```
def
result_validator
(
self
,
func
:
_result
.
ResultValidatorFunc
[
```

AgentDeps
,
ResultData
],
/
)
->
_result
.
ResultValidatorFunc
[
AgentDeps
,
ResultData
]:
"""Decorator to register a result validator function.
Optionally takes [`RunContext`][pydantic_ai.tools.RunContext] as its first argument.
Can decorate a sync or async functions.
Overloads for every possible signature of `result_validator` are included so the decorator doesn't obscure
the type of the function, see `tests/typed_agent.py` for tests.
Example:
```python
from pydantic_ai import Agent, ModelRetry, RunContext
agent = Agent('test', deps_type=str)
@agent.result_validator
def result_validator_simple(data: str) -> str:
if 'wrong' in data:
raise ModelRetry('wrong response')
return data
@agent.result_validator
async def result_validator_deps(ctx: RunContext[str], data: str) -> str:
if ctx.deps in data:
raise ModelRetry('wrong response')
return data
result = agent.run_sync('foobar', deps='spam')
print(result.data)
#> success (no tool calls)
```
"""
self
.
_result_validators
.
append
(
_result
.
ResultValidator
(
func
))
return
func

==
Page: pydantic_ai.models.test - PydanticAI
URL: https://ai.pydantic.dev/api/models/test/
==

pydantic_ai.models.test - PydanticAI

Introduction
API Reference
pydantic_ai.models.test
Utility model for quickly testing apps built with PydanticAI.
TestModel
dataclass
Bases:
Model
A model specifically for testing purposes.
This will (by default) call all tools in the agent, then return a tool response if possible,
otherwise a plain response.
How useful this model is will vary significantly.
Apart from
__init__
derived by the
dataclass
decorator, all methods are private or match those
of the base class.
Source code in

pydantic_ai_slim/pydantic_ai/models/test.py

```
@dataclass
class
TestModel
(
Model
):
"""A model specifically for testing purposes.
This will (by default) call all tools in the agent, then return a tool response if possible,
otherwise a plain response.
How useful this model is will vary significantly.
Apart from `__init__` derived by the `dataclass` decorator, all methods are private or match those
of the base class.
"""
# NOTE: Avoid test discovery by pytest.
__test__
=
False
call_tools
:
list
[
str
]
|
Literal
[
'all'
]
=
'all'
"""List of tools to call. If `'all'`, all tools will be called."""
custom_result_text
:
str
|
None
=
None
"""If set, this text is return as the final result."""
custom_result_args
:
Any
|
None
=
None
"""If set, these args will be passed to the result tool."""
seed
:
int
=
0
"""Seed for generating random data."""
agent_model_function_tools
:
list
[
ToolDefinition
]
|
None
=
field
(
default
=
None
,
init
=
False
)
```

```
"""Definition of function tools passed to the model.
This is set when the model is called, so will reflect the function tools from the last step of the
last run.
"""
agent_model_allow_text_result
:
bool
|
None
=
field
(
default
=
None
,
init
=
False
)
"""Whether plain text responses from the model are allowed.
This is set when the model is called, so will reflect the value from the last step of the last run.
"""
agent_model_result_tools
:
list
[
ToolDefinition
]
|
None
=
field
(
default
=
None
,
init
=
False
)
"""Definition of result tools passed to the model.
This is set when the model is called, so will reflect the result tools from the last step of the last
run.
"""
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
],
)
->
AgentModel
:
self
.
agent_model_function_tools
```

```
=
function_tools
self
.
agent_model_allow_text_result
=
allow_text_result
self
.
agent_model_result_tools
=
result_tools
if
self
.
call_tools
==
'all'
:
tool_calls
=
[(
r
.
name
,
r
)
for
r
in
function_tools
]
else
:
function_tools_lookup
=
{
t
.
name
:
t
for
t
in
function_tools
}
tools_to_call
=
(
function_tools_lookup
[
name
]
for
name
in
self
.
call_tools
)
tool_calls
=
[(
r
.
name
,
r
)
for
r
in
tools_to_call
]
if
```

```
self
.
custom_result_text
is
not
None
:
assert
allow_text_result
,
'Plain response not allowed, but `custom_result_text` is set.'
assert
self
.
custom_result_args
is
None
,
'Cannot set both `custom_result_text` and `custom_result_args`.'
result
:
_utils
.
Either
[
str
|
None
,
Any
|
None
]
=
_utils
.
Either
(
left
=
self
.
custom_result_text
)
elif
self
.
custom_result_args
is
not
None
:
assert
result_tools
is
not
None
,
'No result tools provided, but `custom_result_args` is set.'
result_tool
=
result_tools
[
0
]
if
k
:=
result_tool
.
outer_typed_dict_key
:
result
=
_utils
.
Either
```

```
(
right
=
{
k
:
self
.
custom_result_args
})
else
:
result
=
_utils
.
Either
(
right
=
self
.
custom_result_args
)
elif
allow_text_result
:
result
=
_utils
.
Either
(
left
=
None
)
elif
result_tools
:
result
=
_utils
.
Either
(
right
=
None
)
else
:
result
=
_utils
.
Either
(
left
=
None
)
return
TestAgentModel
(
tool_calls
,
result
,
result_tools
,
self
.
seed
)
def
name
```

(
self
)
->
str
:
return
'test-model'
call_tools
class-attribute
instance-attribute
call_tools
:
list
[
str
]
|
Literal
[
'all'
]
=
'all'
List of tools to call. If
'all'
, all tools will be called.
custom_result_text
class-attribute
instance-attribute
custom_result_text
:
str
|
None
=
None
If set, this text is return as the final result.
custom_result_args
class-attribute
instance-attribute
custom_result_args
:
Any
|
None
=
None
If set, these args will be passed to the result tool.
seed
class-attribute
instance-attribute
seed
:
int
=
0
Seed for generating random data.
agent_model_function_tools
class-attribute
instance-attribute
agent_model_function_tools
:
list
[
ToolDefinition
]
|
None
=
(
field
(
default
=
None
,

init
=
False
)
)
Definition of function tools passed to the model.
This is set when the model is called, so will reflect the function tools from the last step of the last run.
agent_model_allow_text_result
class-attribute
instance-attribute
agent_model_allow_text_result
:
bool
|
None
=
field
(
default
=
None
,
init
=
False
)
Whether plain text responses from the model are allowed.
This is set when the model is called, so will reflect the value from the last step of the last run.
agent_model_result_tools
class-attribute
instance-attribute
agent_model_result_tools
:
list
[
ToolDefinition
]
|
None
=
(
field
(
default
=
None
,
init
=
False
)
)
Definition of result tools passed to the model.
This is set when the model is called, so will reflect the result tools from the last step of the last run.
TestAgentModel
dataclass
Bases:
AgentModel
Implementation of
AgentModel
for testing purposes.
Source code in
pydantic_ai_slim/pydantic_ai/models/test.py

```
@dataclass
class
TestAgentModel
(
AgentModel
):
"""Implementation of `AgentModel` for testing purposes."""
# NOTE: Avoid test discovery by pytest.
__test__
=
False
tool_calls
:
list
[
tuple
[
str
,
ToolDefinition
```

```
]]
# left means the text is plain text; right means it's a function call
result
:
_utils
.
Either
[
str
|
None
,
Any
|
None
]
result_tools
:
list
[
ToolDefinition
]
seed
:
int
async
def
request
(
self
,
messages
:
list
[
Message
])
->
tuple
[
ModelAnyResponse
,
Cost
]:
return
self
.
_request
(
messages
),
Cost
()
@asynccontextmanager
async
def
request_stream
(
self
,
messages
:
list
[
Message
])
->
AsyncIterator
[
EitherStreamedResponse
]:
msg
=
self
.
_request
(
```

```
messages
)
cost
=
Cost
()
if
isinstance
(
msg
,
ModelTextResponse
):
yield
TestStreamTextResponse
(
msg
.
content
,
cost
)
else
:
yield
TestStreamStructuredResponse
(
msg
,
cost
)
def
gen_tool_args
(
self
,
tool_def
:
ToolDefinition
)
->
Any
:
return
_JsonSchemaTestData
(
tool_def
.
parameters_json_schema
,
self
.
seed
)
.
generate
()
def
_request
(
self
,
messages
:
list
[
Message
])
->
ModelAnyResponse
:
# if there are tools, the first thing we want to do is call all of them
if
self
.
tool_calls
and
```

```
not
any
(
m
.
role
==
'model-structured-response'
for
m
in
messages
):
calls
=
[
ToolCall
.
from_dict
(
name
,
self
.
gen_tool_args
(
args
))
for
name
,
args
in
self
.
tool_calls
]
return
ModelStructuredResponse
(
calls
=
calls
)
# get messages since the last model response
new_messages
=
_get_new_messages
(
messages
)
# check if there are any retry prompts, if so retry them
new_retry_names
=
{
m
.
tool_name
for
m
in
new_messages
if
isinstance
(
m
,
RetryPrompt
)}
if
new_retry_names
:
calls
=
[
ToolCall
.
```

```
from_dict
(
name
,
self
.
gen_tool_args
(
args
))
for
name
,
args
in
self
.
tool_calls
if
name
in
new_retry_names
]
return
ModelStructuredResponse
(
calls
=
calls
)
if
response_text
:=
self
.
result
.
left
:
if
response_text
.
value
is
None
:
# build up details of tool responses
output
:
dict
[
str
,
Any
]
=
{}
for
message
in
messages
:
if
isinstance
(
message
,
ToolReturn
):
output
[
message
.
tool_name
]
=
message
```

```
.
content
if
output
:
return
ModelTextResponse
(
content
=
pydantic_core
.
to_json
(
output
)
.
decode
())
else
:
return
ModelTextResponse
(
content
=
'success (no tool calls)'
)
else
:
return
ModelTextResponse
(
content
=
response_text
.
value
)
else
:
assert
self
.
result_tools
,
'No result tools provided'
custom_result_args
=
self
.
result
.
right
result_tool
=
self
.
result_tools
[
self
.
seed
%
len
(
self
.
result_tools
)]
if
custom_result_args
is
not
None
:
return
```

```
ModelStructuredResponse
(
calls
=
[
ToolCall
.
from_dict
(
result_tool
.
name
,
custom_result_args
)])
else
:
response_args
=
self
.
gen_tool_args
(
result_tool
)
return
ModelStructuredResponse
(
calls
=
[
ToolCall
.
from_dict
(
result_tool
.
name
,
response_args
)])
```

TestStreamTextResponse

dataclass

Bases:
StreamTextResponse

A text response that streams test data.

Source code in
pydantic_ai_slim/pydantic_ai/models/test.py

```
@dataclass
class TestStreamTextResponse(StreamTextResponse):
    """A text response that streams test data."""
    _text: str
    _cost: Cost
    _iter: Iterator[str] = field(init=False)
    _timestamp: datetime = field(default_factory=_utils.now_utc)
    _buffer: list[str] = field(default_factory=list, init=False)
    def __post_init__(self):
        *words, last_word = self._text.split(' ')
        words =
```

```
[
f
'
{
word
}
'
for
word
in
words
]
words
.
append
(
last_word
)
if
len
(
words
)
==
1
and
len
(
self
.
_text
)
>
2
:
mid
=
len
(
self
.
_text
)
//
2
words
=
[
self
.
_text
[:
mid
],
self
.
_text
[
mid
:]]
self
.
_iter
=
iter
(
words
)
async
def
__anext__
(
self
)
->
None
:
```

```
self
.
_buffer
.
append
(
_utils
.
sync_anext
(
self
.
_iter
))
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
Iterable
[
str
]:
yield from
self
.
_buffer
self
.
_buffer
.
clear
()
def
cost
(
self
)
->
Cost
:
return
self
.
_cost
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp
```

TestStreamStructuredResponse
dataclass
Bases:
StreamStructuredResponse
A structured response that streams test data.
Source code in
pydantic_ai_slim/pydantic_ai/models/test.py

```python
@dataclass
class TestStreamStructuredResponse(StreamStructuredResponse):
    """A structured response that streams test data."""
    _structured_response: ModelStructuredResponse
    _cost: Cost
    _iter: Iterator[None] = field(default_factory=lambda: iter([None]))
    _timestamp: datetime = field(default_factory=_utils.now_utc, init=False)

    async def __anext__(self) -> None:
        return _utils.sync_anext(self
```

```
.
_iter
)
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
ModelStructuredResponse
:
return
self
.
_structured_response
def
cost
(
self
)
->
Cost
:
return
self
.
_cost
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp
```


==
Page: Stream whales - PydanticAI
URL: https://ai.pydantic.dev/examples/stream-whales/
==

Stream whales - PydanticAI

Pydantic Model
Weather agent
Bank support
SQL Generation
RAG
Stream markdown
Stream whales
Stream whales
Table of contents
Running the Example
Example Code
Chat App with FastAPI
API Reference
API Reference
pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Running the Example
Example Code
Introduction
Examples
Stream whales
Information about whales — an example of streamed structured response validation.
Demonstrates:
streaming structured responses
This script streams structured responses from GPT-4 about whales, validates the data and displays it as a dynamic table using
rich
as the data is received.
Running the Example
With
dependencies installed and environment variables set
, run:
pip
uv
python
-m
pydantic_ai_examples.stream_whales
uv
run
-m
pydantic_ai_examples.stream_whales
Should give an output like this:
Example Code
stream_whales.py
from
typing
import
Annotated
,
NotRequired
,
TypedDict
import
devtools
import
logfire
from
pydantic
import
Field
,
ValidationError
from
rich.console

```
import
Console
from
rich.live
import
Live
from
rich.table
import
Table
from
pydantic_ai
import
Agent
# 'if-token-present' means nothing will be sent (and the example will work) if you don't have
logfire configured
logfire
.
configure
(
send_to_logfire
=
'if-token-present'
)
class
Whale
(
TypedDict
):
name
:
str
length
:
Annotated
[
float
,
Field
(
description
=
'Average length of an adult whale in meters.'
)
]
weight
:
NotRequired
[
Annotated
[
float
,
Field
(
description
=
'Average weight of an adult whale in kilograms.'
,
ge
=
50
),
]
]
ocean
:
NotRequired
[
str
]
description
:
NotRequired
[
Annotated
[
```

```
str
,
Field
(
description
=
'Short Description'
)]]
agent
=
Agent
(
'openai:gpt-4'
,
result_type
=
list
[
Whale
])
def
check_validation_error
(
e
:
ValidationError
)
->
bool
:
devtools
.
debug
(
e
.
errors
())
return
False
async
def
main
():
console
=
Console
()
with
Live
(
'
\n
'
*
36
,
console
=
console
)
as
live
:
console
.
print
(
'Requesting data...'
,
style
=
'cyan'
)
async
with
agent
```

```
.
run_stream
(
'Generate me details of 5 species of Whale.'
)
as
result
:
console
.
print
(
'Response:'
,
style
=
'green'
)
async
for
message
,
last
in
result
.
stream_structured
(
debounce_by
=
0.01
):
try
:
whales
=
await
result
.
validate_structured_result
(
message
,
allow_partial
=
not
last
)
except
ValidationError
as
exc
:
if
all
(
e
[
'type'
]
==
'missing'
and
e
[
'loc'
]
==
(
'response'
,)
for
e
in
exc
.
errors
```

```
()
):
continue
else
:
raise
table
=
Table
(
title
=
'Species of Whale'
,
caption
=
'Streaming Structured responses from GPT-4'
,
width
=
120
,
)
table
.
add_column
(
'ID'
,
justify
=
'right'
)
table
.
add_column
(
'Name'
)
table
.
add_column
(
'Avg. Length (m)'
,
justify
=
'right'
)
table
.
add_column
(
'Avg. Weight (kg)'
,
justify
=
'right'
)
table
.
add_column
(
'Ocean'
)
table
.
add_column
(
'Description'
,
justify
=
'right'
)
for
wid
```

```
,
whale
in
enumerate
(
whales
,
start
=
1
):
table
.
add_row
(
str
(
wid
),
whale
[
'name'
],
f
'
{
whale
[
"length"
]
:
0.0f
}
'
,
f
'
{
w
:
0.0f
}
'
if
(
w
:=
whale
.
get
(
'weight'
))
else
'…'
,
whale
.
get
(
'ocean'
)
or
'…'
,
whale
.
get
(
'description'
)
or
'…'
,
)
live
.
```

```
update
(
table
)
if
__name__
==
'__main__'
:
import
asyncio
asyncio
.
run
(
main
())
```


==
Page: pydantic_ai.models.function - PydanticAI
URL: https://ai.pydantic.dev/api/models/function/
==

pydantic_ai.models.function - PydanticAI

Table of contents

Table of contents
function
FunctionModel
__init__
AgentInfo
function_tools
allow_text_result
result_tools
DeltaToolCall
name
json_args
DeltaToolCalls
FunctionDef
StreamFunctionDef
FunctionAgentModel
FunctionStreamTextResponse
FunctionStreamStructuredResponse
Introduction
API Reference

pydantic_ai.models.function

A model controlled by a local function.

FunctionModel
is similar to
TestModel
,
but allows greater control over the model's behavior.
Its primary use case is for more advanced unit testing than is possible with
TestModel
.

FunctionModel
dataclass

Bases:
Model

A model controlled by a local function.

Apart from
__init__
, all methods are private or match those of the base class.

Source code in
pydantic_ai_slim/pydantic_ai/models/function.py

```
@dataclass(init=False)
class FunctionModel(Model):
    """A model controlled by a local function.
    Apart from `__init__`, all methods are private or match those of the base class.
    """
    function: FunctionDef | None = None
    stream_function: StreamFunctionDef | None = None

    @overload
    def __init__(self, function: FunctionDef) -> None: ...

    @overload
    def __init__(self, *, stream_function:
```

```
StreamFunctionDef
)
->
None
:
...
@overload
def
__init__
(
self
,
function
:
FunctionDef
,
*
,
stream_function
:
StreamFunctionDef
)
->
None
:
...
def
__init__
(
self
,
function
:
FunctionDef
|
None
=
None
,
*
,
stream_function
:
StreamFunctionDef
|
None
=
None
):
"""Initialize a `FunctionModel`.
Either `function` or `stream_function` must be provided, providing both is allowed.
Args:
function: The function to call for non-streamed requests.
stream_function: The function to call for streamed requests.
"""
if
function
is
None
and
stream_function
is
None
:
raise
TypeError
(
'Either `function` or `stream_function` must be provided'
)
self
.
function
=
function
self
.
stream_function
```

```
=
stream_function
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
],
)
->
AgentModel
:
return
FunctionAgentModel
(
self
.
function
,
self
.
stream_function
,
AgentInfo
(
function_tools
,
allow_text_result
,
result_tools
)
)
def
name
(
self
)
->
str
:
labels
:
list
[
str
]
=
[]
if
self
.
function
is
not
None
:
labels
.
append
```

```
(
self
.
function
.
__name__
)
if
self
.
stream_function
is
not
None
:
labels
.
append
(
f
'stream-
{
self
.
stream_function
.
__name__
}
'
)
return
f
'function:
{
","
.
join
(
labels
)
}
'
__init__
__init__
(
function
:
FunctionDef
)
->
None
__init__
(
*
,
stream_function
:
StreamFunctionDef
)
->
None
__init__
(
function
:
FunctionDef
,
*
,
stream_function
:
StreamFunctionDef
)
->
None
__init__
(
```

function
:
FunctionDef
|
None
=
None
,
*
,
stream_function
:
StreamFunctionDef
|
None
=
None
)
Initialize a
FunctionModel
.
Either
function
or
stream_function
must be provided, providing both is allowed.
Parameters:

Name	Type	Description	Default
function	FunctionDef | None	The function to call for non-streamed requests.	None
stream_function	StreamFunctionDef | None	The function to call for streamed requests.	None

Source code in
pydantic_ai_slim/pydantic_ai/models/function.py

```
def
__init__
(
self
,
function
:
FunctionDef
|
None
=
None
,
*
,
stream_function
:
StreamFunctionDef
|
None
=
```

```
None
):
"""Initialize a `FunctionModel`.
Either `function` or `stream_function` must be provided, providing both is allowed.
Args:
function: The function to call for non-streamed requests.
stream_function: The function to call for streamed requests.
"""
if
function
is
None
and
stream_function
is
None
:
raise
TypeError
(
'Either `function` or `stream_function` must be provided'
)
self
.
function
=
function
self
.
stream_function
=
stream_function
```

AgentInfo
dataclass
Information about an agent.
This is passed as the second to functions used within
FunctionModel
.
Source code in
pydantic_ai_slim/pydantic_ai/models/function.py

```
@dataclass
(
frozen
=
True
)
class
AgentInfo
:
"""Information about an agent.
This is passed as the second to functions used within [`FunctionModel`]
[pydantic_ai.models.function.FunctionModel].
"""
function_tools
:
list
[
ToolDefinition
]
"""The function tools available on this agent.
```

These are the tools registered via the [`tool`][pydantic_ai.Agent.tool] and
[`tool_plain`][pydantic_ai.Agent.tool_plain] decorators.
"""
allow_text_result
:
bool
"""Whether a plain text result is allowed."""
result_tools
:
list
[
ToolDefinition
]
"""The tools that can called as the final result of the run."""
function_tools
instance-attribute
function_tools
:
list
[
ToolDefinition
]
The function tools available on this agent.
These are the tools registered via the
tool
and
tool_plain
decorators.
allow_text_result
instance-attribute
allow_text_result
:
bool
Whether a plain text result is allowed.
result_tools
instance-attribute
result_tools
:
list
[
ToolDefinition
]
The tools that can called as the final result of the run.
DeltaToolCall
dataclass
Incremental change to a tool call.
Used to describe a chunk when streaming structured responses.
Source code in
pydantic_ai_slim/pydantic_ai/models/function.py
@dataclass
class
DeltaToolCall
:
"""Incremental change to a tool call.
Used to describe a chunk when streaming structured responses.
"""
name
:
str
|
None
=
None
"""Incremental change to the name of the tool."""
json_args
:

str
|
None
=
None
"""Incremental change to the arguments as JSON"""
name
class-attribute
instance-attribute
name
:
str
|
None
=
None
Incremental change to the name of the tool.
json_args
class-attribute
instance-attribute
json_args
:
str
|
None
=
None
Incremental change to the arguments as JSON
DeltaToolCalls
module-attribute
DeltaToolCalls
:
TypeAlias
=
dict
[
int
,
DeltaToolCall
]
A mapping of tool call IDs to incremental changes.
FunctionDef
module-attribute
FunctionDef
:
TypeAlias
=
Callable
[
[
list
[
Message
],
AgentInfo
],
Union
[
ModelAnyResponse
,
Awaitable
[
ModelAnyResponse
]],
]
A function used to generate a non-streamed response.
StreamFunctionDef
module-attribute
StreamFunctionDef
:
TypeAlias
=
Callable
[
[
list
[

Message
],
AgentInfo
],
AsyncIterator
[
Union
[
str
,
DeltaToolCalls
]],
]
A function used to generate a streamed response.
While this is defined as having return type of
AsyncIterator[Union[str, DeltaToolCalls]]
, it should
really be considered as
Union[AsyncIterator[str], AsyncIterator[DeltaToolCalls]
,
E.g. you need to yield all text or all
DeltaToolCalls
, not mix them.
FunctionAgentModel
dataclass
Bases:
AgentModel
Implementation of
AgentModel
for
FunctionModel
.
Source code in
pydantic_ai_slim/pydantic_ai/models/function.py

```
@dataclass
class
FunctionAgentModel
(
AgentModel
):
"""Implementation of `AgentModel` for [FunctionModel][pydantic_ai.models.function.FunctionModel]."""
function
```

```
:
FunctionDef
|
None
stream_function
:
StreamFunctionDef
|
None
agent_info
:
AgentInfo
async
def
request
(
self
,
messages
:
list
[
Message
])
->
tuple
[
ModelAnyResponse
,
result
.
Cost
]:
assert
self
.
function
is
not
None
,
'FunctionModel must receive a `function` to support non-streamed requests'
if
inspect
.
iscoroutinefunction
(
self
.
function
):
response
=
await
self
.
function
(
messages
,
self
.
agent_info
)
else
:
response_
=
await
_utils
.
run_in_executor
(
self
.
function
,
```

```
messages
,
self
.
agent_info
)
response
=
cast
(
ModelAnyResponse
,
response_
)
# TODO is `messages` right here? Should it just be new messages?
return
response
,
_estimate_cost
(
chain
(
messages
,
[
response
]))
@asynccontextmanager
async
def
request_stream
(
self
,
messages
:
list
[
Message
])
->
AsyncIterator
[
EitherStreamedResponse
]:
assert
(
self
.
stream_function
is
not
None
),
'FunctionModel must receive a `stream_function` to support streamed requests'
response_stream
=
self
.
stream_function
(
messages
,
self
.
agent_info
)
try
:
first
=
await
response_stream
.
__anext__
()
except
```

```
StopAsyncIteration
as
e
:
raise
ValueError
(
'Stream function must return at least one item'
)
from
e
if
isinstance
(
first
,
str
):
text_stream
=
cast
(
AsyncIterator
[
str
],
response_stream
)
yield
FunctionStreamTextResponse
(
first
,
text_stream
)
else
:
structured_stream
=
cast
(
AsyncIterator
[
DeltaToolCalls
],
response_stream
)
yield
FunctionStreamStructuredResponse
(
first
,
structured_stream
)
```

FunctionStreamTextResponse
dataclass
Bases:
StreamTextResponse
Implementation of
StreamTextResponse
for
FunctionModel
.
Source code in
pydantic_ai_slim/pydantic_ai/models/function.py

```
@dataclass
class
FunctionStreamTextResponse
(
StreamTextResponse
):
"""Implementation of `StreamTextResponse` for [FunctionModel]
[pydantic_ai.models.function.FunctionModel]."""
_next
:
str
|
None
_iter
:
AsyncIterator
[
str
]
_timestamp
:
datetime
=
field
(
default_factory
=
_utils
.
now_utc
,
init
=
False
)
_buffer
:
list
[
str
]
=
field
(
default_factory
=
list
,
init
=
False
)
async
def
__anext__
(
self
)
->
None
:
if
self
.
```

```
_next
is
not
None
:
self
.
_buffer
.
append
(
self
.
_next
)
self
.
_next
=
None
else
:
self
.
_buffer
.
append
(
await
self
.
_iter
.
__anext__
())
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
Iterable
[
str
]:
yield from
self
.
_buffer
self
.
_buffer
.
clear
()
def
cost
(
self
)
->
result
.
Cost
:
return
result
.
Cost
```

()
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp

FunctionStreamStructuredResponse
dataclass
Bases:
StreamStructuredResponse
Implementation of
StreamStructuredResponse
for
FunctionModel
.
Source code in
pydantic_ai_slim/pydantic_ai/models/function.py

```
@dataclass
class
FunctionStreamStructuredResponse
(
StreamStructuredResponse
):
"""Implementation of `StreamStructuredResponse` for [FunctionModel]
[pydantic_ai.models.function.FunctionModel]."""
_next
:
DeltaToolCalls
|
None
_iter
:
AsyncIterator
[
```

```
DeltaToolCalls
]
_delta_tool_calls
:
dict
[
int
,
DeltaToolCall
]
=
field
(
default_factory
=
dict
)
_timestamp
:
datetime
=
field
(
default_factory
=
_utils
.
now_utc
)
async
def
__anext__
(
self
)
->
None
:
if
self
.
_next
is
not
None
:
tool_call
=
self
.
_next
self
.
_next
=
None
else
:
tool_call
=
await
self
.
_iter
.
__anext__
()
for
key
,
new
in
tool_call
.
items
():
if
```

```
current
:=
self
.
_delta_tool_calls
.
get
(
key
):
current
.
name
=
_utils
.
add_optional
(
current
.
name
,
new
.
name
)
current
.
json_args
=
_utils
.
add_optional
(
current
.
json_args
,
new
.
json_args
)
else
:
self
.
_delta_tool_calls
[
key
]
=
new
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
ModelStructuredResponse
:
calls
:
list
[
ToolCall
]
=
[]
for
```

```
c
in
self
.
_delta_tool_calls
.
values
():
if
c
.
name
is
not
None
and
c
.
json_args
is
not
None
:
calls
.
append
(
ToolCall
.
from_json
(
c
.
name
,
c
.
json_args
))
return
ModelStructuredResponse
(
calls
,
timestamp
=
self
.
_timestamp
)
def
cost
(
self
)
->
result
.
Cost
:
return
result
.
Cost
()
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp
```

==
Page: Testing and Evals - PydanticAI
URL: https://ai.pydantic.dev/testing-evals/
==

Testing and Evals - PydanticAI
Skip to content
PydanticAI
Testing and Evals
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

Introduction
Documentation
Testing and Evals
With PydanticAI and LLM integrations in general, there are two distinct kinds of test:
Unit tests
— tests of your application code, and whether it's behaving correctly

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

```
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
(
# (3)!
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
)
```

)
DatabaseConn
is a class that holds a database connection
WeatherService
has methods to get weather forecasts and historic data about the weather
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
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

```
from
datetime
import
timezone
import
pytest
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'
,
),
]
```

We're using
anyio
to run async tests.
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.
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.
Now we call the function we want to test inside the
override
context manager.
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
.
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.
The
IsNow
helper allows us to use declarative asserts even with data which will contain timestamps that change over time.
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

```
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
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'
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.
Our function is slightly intelligent in that it tries to extract a date from the prompt, but just hard codes the location.
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

```
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

```
import
json
from
pathlib
import
Path
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)
```

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.

The `build_prompt` method constructs the system prompt from the examples and the database type.

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.

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

```
{
"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

```
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
():
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.

==
Page: pydantic_ai.models - PydanticAI
URL: https://ai.pydantic.dev/api/models/base/
==

pydantic_ai.models - PydanticAI

Table of contents
models
KnownModelName
Model
agent_model
AgentModel
request
request_stream
StreamTextResponse
__aiter__
__anext__
get
cost
timestamp
StreamStructuredResponse
__aiter__
__anext__
get
cost
timestamp
ALLOW_MODEL_REQUESTS
check_allow_model_requests
override_allow_model_requests

pydantic_ai.models
Logic related to making requests to an LLM.
The aim here is to make a common interface for different LLMs, so that the rest of the code can be agnostic to the
specific LLM being used.
KnownModelName
module-attribute
```
KnownModelName
=
Literal
[
"openai:gpt-4o"
,
"openai:gpt-4o-mini"
,
"openai:gpt-4-turbo"
,
"openai:gpt-4"
,
"openai:o1-preview"
,
"openai:o1-mini"
```

```
,
"openai:gpt-3.5-turbo"
,
"groq:llama-3.3-70b-versatile"
,
"groq:llama-3.1-70b-versatile"
,
"groq:llama3-groq-70b-8192-tool-use-preview"
,
"groq:llama3-groq-8b-8192-tool-use-preview"
,
"groq:llama-3.1-70b-specdec"
,
"groq:llama-3.1-8b-instant"
,
"groq:llama-3.2-1b-preview"
,
"groq:llama-3.2-3b-preview"
,
"groq:llama-3.2-11b-vision-preview"
,
"groq:llama-3.2-90b-vision-preview"
,
"groq:llama3-70b-8192"
,
"groq:llama3-8b-8192"
,
"groq:mixtral-8x7b-32768"
,
"groq:gemma2-9b-it"
,
"groq:gemma-7b-it"
,
"gemini-1.5-flash"
,
"gemini-1.5-pro"
,
"vertexai:gemini-1.5-flash"
,
"vertexai:gemini-1.5-pro"
,
"ollama:codellama"
,
"ollama:gemma"
,
"ollama:gemma2"
,
"ollama:llama3"
,
"ollama:llama3.1"
,
"ollama:llama3.2"
,
"ollama:llama3.2-vision"
,
"ollama:llama3.3"
,
"ollama:mistral"
,
"ollama:mistral-nemo"
,
"ollama:mixtral"
,
"ollama:phi3"
,
"ollama:qwq"
,
"ollama:qwen"
,
"ollama:qwen2"
,
"ollama:qwen2.5"
,
"ollama:starcoder2"
,
"claude-3-5-haiku-latest"
,
```

```
"claude-3-5-sonnet-latest"
,
"claude-3-opus-latest"
,
"test"
,
]
```

Known model names that can be used with the
model
parameter of
Agent
.
KnownModelName
is provided as a concise way to specify a model.

Model

Bases:
ABC

Abstract class for a model.

Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```
class
Model
(
ABC
):
"""Abstract class for a model."""
@abstractmethod
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
```

ToolDefinition
],
)
->
AgentModel
:
"""Create an agent model, this is called for each step of an agent run.
This is async in case slow/async config checks need to be performed that can't be done in
`__init__`.
Args:
function_tools: The tools available to the agent.
allow_text_result: Whether a plain text final response/result is permitted.
result_tools: Tool definitions for the final result tool(s), if any.
Returns:
An agent model.
"""
raise
NotImplementedError
()
@abstractmethod
def
name
(
self
)
->
str
:
raise
NotImplementedError
()
agent_model
abstractmethod
async
agent_model
(
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
]
)
->
AgentModel
Create an agent model, this is called for each step of an agent run.
This is async in case slow/async config checks need to be performed that can't be done in
__init__
.
Parameters:
Name
Type
Description
Default
function_tools
list
[
ToolDefinition
]
The tools available to the agent.
required
allow_text_result
bool
Whether a plain text final response/result is permitted.
required

result_tools
list
[
ToolDefinition
]
Tool definitions for the final result tool(s), if any.
required

Returns:

Type
Description
AgentModel
An agent model.

Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```
@abstractmethod
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
],
)
->
AgentModel
:
"""Create an agent model, this is called for each step of an agent run.
This is async in case slow/async config checks need to be performed that can't be done in
`__init__`.
Args:
function_tools: The tools available to the agent.
allow_text_result: Whether a plain text final response/result is permitted.
result_tools: Tool definitions for the final result tool(s), if any.
Returns:
An agent model.
"""
raise
NotImplementedError
()
```

AgentModel

Bases: ABC

Model configured for each step of an Agent run.

Source code in pydantic_ai_slim/pydantic_ai/models/__init__.py

```python
class AgentModel(ABC):
    """Model configured for each step of an Agent run."""

    @abstractmethod
    async def request(self, messages: list[Message]) -> tuple[ModelAnyResponse, Cost]:
        """Make a request to the model."""
        raise NotImplementedError()

    @asynccontextmanager
    async def request_stream(self, messages: list[Message]) -> AsyncIterator[EitherStreamedResponse]:
        """Make a request to the model and return a streaming response."""
        raise NotImplementedError(f'Streamed requests not supported by this {self
```

```
.
__class__
.
__name__
}
'
)
# yield is required to make this a generator for type checking
# noinspection PyUnreachableCode
yield
# pragma: no cover
request
abstractmethod
async
request
(
messages
:
list
[
Message
],
)
->
tuple
[
ModelAnyResponse
,
Cost
]
```

Make a request to the model.

Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```
@abstractmethod
async
def
request
(
self
,
messages
:
list
[
Message
])
->
tuple
[
ModelAnyResponse
,
Cost
]:
"""Make a request to the model."""
raise
NotImplementedError
()
request_stream
async
request_stream
(
messages
:
list
[
Message
],
)
->
AsyncIterator
[
EitherStreamedResponse
]
```

Make a request to the model and return a streaming response.
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```
@asynccontextmanager
async
def
request_stream
(
self
,
messages
:
list
[
Message
])
->
AsyncIterator
[
EitherStreamedResponse
]:
"""Make a request to the model and return a streaming response."""
raise
NotImplementedError
(
f
'Streamed requests not supported by this
{
self
.
__class__
.
__name__
}
'
)
# yield is required to make this a generator for type checking
# noinspection PyUnreachableCode
yield
# pragma: no cover
```

StreamTextResponse
Bases:
ABC
Streamed response from an LLM when returning text.
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```python
class StreamTextResponse(ABC):
    """Streamed response from an LLM when returning text."""

    def __aiter__(self) -> AsyncIterator[None]:
        """Stream the response as an async iterable, building up the text as it goes.

        This is an async iterator that yields `None` to avoid doing the work of validating the input and
        extracting the text field when it will often be thrown away.
        """
        return self

    @abstractmethod
    async def __anext__(self) -> None:
        """Process the next chunk of the response, see above for why this returns `None`."""
        raise NotImplementedError()

    @abstractmethod
    def get(self, *, final: bool = False) -> Iterable[str]:
        """Returns an iterable of text since the last call to `get()` — e.g. the text delta.

        Args:
            final: If True, this is the final call, after iteration is complete, the response should be fully validated
                and all text extracted.
        """
        raise NotImplementedError
```

()
@abstractmethod
def
cost
(
self
)
->
Cost
:
"""Return the cost of the request.
NOTE: this won't return the ful cost until the stream is finished.
"""
raise
NotImplementedError
()
@abstractmethod
def
timestamp
(
self
)
->
datetime
:
"""Get the timestamp of the response."""
raise
NotImplementedError
()
__aiter__
__aiter__
()
->
AsyncIterator
[
None
]
Stream the response as an async iterable, building up the text as it goes.
This is an async iterator that yields
None
to avoid doing the work of validating the input and
extracting the text field when it will often be thrown away.
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py
def
__aiter__
(
self
)
->
AsyncIterator
[
None
]:
"""Stream the response as an async iterable, building up the text as it goes.
This is an async iterator that yields `None` to avoid doing the work of validating the input and
extracting the text field when it will often be thrown away.
"""
return
self
__anext__
abstractmethod
async
__anext__
()
->
None
Process the next chunk of the response, see above for why this returns
None
.

Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```
@abstractmethod
async
def
__anext__
(
self
)
->
None
:
"""Process the next chunk of the response, see above for why this returns `None`."""
raise
NotImplementedError
()
```

get
abstractmethod
get
(
*
,
final
:
bool
=
False
)
->
Iterable
[
str
]

Returns an iterable of text since the last call to
get()
— e.g. the text delta.

Parameters:

Name	Type	Description	Default
final	bool	If True, this is the final call, after iteration is complete, the response should be fully validated and all text extracted.	False

Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```
@abstractmethod
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
Iterable
```

[
str
]:
"""Returns an iterable of text since the last call to `get()` — e.g. the text delta.
Args:
final: If True, this is the final call, after iteration is complete, the response should be fully validated
and all text extracted.
"""
raise
NotImplementedError
()
cost
abstractmethod
cost
()
->
Cost
Return the cost of the request.
NOTE: this won't return the ful cost until the stream is finished.
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py
@abstractmethod
def
cost
(
self
)
->
Cost
:
"""Return the cost of the request.
NOTE: this won't return the ful cost until the stream is finished.
"""
raise
NotImplementedError
()
timestamp
abstractmethod
timestamp
()
->
datetime
Get the timestamp of the response.
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py
@abstractmethod
def
timestamp
(
self
)
->
datetime
:
"""Get the timestamp of the response."""
raise
NotImplementedError
()
StreamStructuredResponse
Bases:
ABC
Streamed response from an LLM when calling a tool.
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```python
class StreamStructuredResponse(ABC):
    """Streamed response from an LLM when calling a tool."""
    def __aiter__(self) -> AsyncIterator[None]:
        """Stream the response as an async iterable, building up the tool call as it goes.
        This is an async iterator that yields `None` to avoid doing the work of building the final tool call when
        it will often be thrown away.
        """
        return self
    @abstractmethod
    async def __anext__(self) -> None:
        """Process the next chunk of the response, see above for why this returns `None`."""
        raise NotImplementedError()
    @abstractmethod
    def
```

get
(
self
,
*
,
final
:
bool
=
False
)
->
ModelStructuredResponse
:
"""Get the `ModelStructuredResponse` at this point.
The `ModelStructuredResponse` may or may not be complete, depending on whether the stream is finished.
Args:
final: If True, this is the final call, after iteration is complete, the response should be fully validated.
"""
raise
NotImplementedError
()
@abstractmethod
def
cost
(
self
)
->
Cost
:
"""Get the cost of the request.
NOTE: this won't return the full cost until the stream is finished.
"""
raise
NotImplementedError
()
@abstractmethod
def
timestamp
(
self
)
->
datetime
:
"""Get the timestamp of the response."""
raise
NotImplementedError
()
__aiter__
__aiter__
()
->
AsyncIterator
[
None
]
Stream the response as an async iterable, building up the tool call as it goes.
This is an async iterator that yields
None
to avoid doing the work of building the final tool call when
it will often be thrown away.
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py
def
__aiter__

(
self
)
->
AsyncIterator
[
None
]:
"""Stream the response as an async iterable, building up the tool call as it goes.
This is an async iterator that yields `None` to avoid doing the work of building the final tool call when
it will often be thrown away.
"""
return
self
__anext__
abstractmethod
async
__anext__
()
->
None
Process the next chunk of the response, see above for why this returns
None
.
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py
@abstractmethod
async
def
__anext__
(
self
)
->
None
:
"""Process the next chunk of the response, see above for why this returns `None`."""
raise
NotImplementedError
()
get
abstractmethod
get
(
*
,
final
:
bool
=
False
)
->
ModelStructuredResponse
Get the
ModelStructuredResponse
at this point.
The
ModelStructuredResponse
may or may not be complete, depending on whether the stream is finished.
Parameters:
Name
Type
Description
Default
final
bool
If True, this is the final call, after iteration is complete, the response should be fully validated.
False
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```
@abstractmethod
def get(self, *, final: bool = False) -> ModelStructuredResponse:
    """Get the `ModelStructuredResponse` at this point.
    The `ModelStructuredResponse` may or may not be complete, depending on whether the stream is finished.
    Args:
    final: If True, this is the final call, after iteration is complete, the response should be fully validated.
    """
    raise NotImplementedError()
```

cost
abstractmethod
cost() -> Cost

Get the cost of the request.

NOTE: this won't return the full cost until the stream is finished.

Source code in pydantic_ai_slim/pydantic_ai/models/__init__.py

```
@abstractmethod
def cost(self) -> Cost:
    """Get the cost of the request.
    NOTE: this won't return the full cost until the stream is finished.
    """
    raise NotImplementedError()
```

timestamp
abstractmethod
timestamp() -> datetime

Get the timestamp of the response.

Source code in

pydantic_ai_slim/pydantic_ai/models/__init__.py
@abstractmethod
def
timestamp
(
self
)
->
datetime
:
"""Get the timestamp of the response."""
raise
NotImplementedError
()
ALLOW_MODEL_REQUESTS
module-attribute
ALLOW_MODEL_REQUESTS
=
True
Whether to allow requests to models.
This global setting allows you to disable request to most models, e.g. to make sure you don't accidentally
make costly requests to a model during tests.
The testing models
TestModel
and
FunctionModel
are no affected by this setting.
check_allow_model_requests
check_allow_model_requests
()
->
None
Check if model requests are allowed.
If you're defining your own models that have cost or latency associated with their use, you should call this in
Model.agent_model
.
Raises:
Type
Description
RuntimeError
If model requests are not allowed.
Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py
def
check_allow_model_requests
()
->
None
:
"""Check if model requests are allowed.
If you're defining your own models that have cost or latency associated with their use, you should call this in
[`Model.agent_model`][pydantic_ai.models.Model.agent_model].
Raises:
RuntimeError: If model requests are not allowed.
"""
if
not
ALLOW_MODEL_REQUESTS
:

```
raise
RuntimeError
(
'Model requests are not allowed, since ALLOW_MODEL_REQUESTS is False'
)
```

override_allow_model_requests

```
override_allow_model_requests
(
allow_model_requests
:
bool
,
)
->
Iterator
[
None
]
```

Context manager to temporarily override
ALLOW_MODEL_REQUESTS
.

Parameters:

Name	Type	Description	Default
allow_model_requests	bool	Whether to allow model requests within the context.	required

Source code in
pydantic_ai_slim/pydantic_ai/models/__init__.py

```
@contextmanager
def
override_allow_model_requests
(
allow_model_requests
:
bool
)
->
Iterator
[
None
]:
"""Context manager to temporarily override [`ALLOW_MODEL_REQUESTS`]
[pydantic_ai.models.ALLOW_MODEL_REQUESTS].
Args:
allow_model_requests: Whether to allow model requests within the context.
"""
global
ALLOW_MODEL_REQUESTS
old_value
=
ALLOW_MODEL_REQUESTS
ALLOW_MODEL_REQUESTS
=
allow_model_requests
# pyright: ignore[reportConstantRedefinition]
try
:
yield
finally
```

:
ALLOW_MODEL_REQUESTS
=
old_value
pyright: ignore[reportConstantRedefinition]

==
Page: pydantic_ai.result - PydanticAI
URL: https://ai.pydantic.dev/api/result/
==

pydantic_ai.result - PydanticAI
Skip to content
PydanticAI
pydantic_ai.result
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

total_tokens
details
__add__
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
result
ResultData
RunResult
all_messages
all_messages_json
new_messages
new_messages_json
data
cost
StreamedRunResult
all_messages
all_messages_json
new_messages
new_messages_json
cost_so_far
is_complete
stream
stream_text
stream_structured
get_data
is_structured
cost
timestamp
validate_structured_result
Cost
request_tokens
response_tokens
total_tokens
details
__add__
Introduction
API Reference
pydantic_ai.result
ResultData
module-attribute
ResultData
=
TypeVar
(
'ResultData'
)
Type variable for the result data of a run.
RunResult
dataclass
Bases:
_BaseRunResult
[
ResultData
]
Result of a non-streamed run.
Source code in
pydantic_ai_slim/pydantic_ai/result.py

@dataclass
class
RunResult
(
_BaseRunResult
[
ResultData
]):
"""Result of a non-streamed run."""
data
:
ResultData
"""Data from the final response in the run."""
_cost
:
Cost
def
cost
(
self
)
->
Cost
:
"""Return the cost of the whole run."""
return
self
.
_cost
all_messages
all_messages
()
->
list
[
Message
]
Return the history of messages.
Source code in
pydantic_ai_slim/pydantic_ai/result.py
def
all_messages
(
self
)
->
list
[
messages
.
Message
]:
"""Return the history of messages."""
this is a method to be consistent with the other methods
return
self
.
_all_messages
all_messages_json
all_messages_json
()
->
bytes
Return all messages from
all_messages
as JSON bytes.
Source code in
pydantic_ai_slim/pydantic_ai/result.py
def

all_messages_json
(
self
)
->
bytes
:
"""Return all messages from [`all_messages`][..all_messages] as JSON bytes."""
return
messages
.
MessagesTypeAdapter
.
dump_json
(
self
.
all_messages
())
new_messages
new_messages
()
->
list
[
Message
]
Return new messages associated with this run.
System prompts and any messages from older runs are excluded.
Source code in
pydantic_ai_slim/pydantic_ai/result.py
def
new_messages
(
self
)
->
list
[
messages
.
Message
]:
"""Return new messages associated with this run.
System prompts and any messages from older runs are excluded.
"""
return
self
.
all_messages
()[
self
.
_new_message_index
:]
new_messages_json
new_messages_json
()
->
bytes
Return new messages from
new_messages
as JSON bytes.
Source code in
pydantic_ai_slim/pydantic_ai/result.py
def
new_messages_json
(

self
)
->
bytes
:
"""Return new messages from [`new_messages`][..new_messages] as JSON bytes."""
return
messages
.
MessagesTypeAdapter
.
dump_json
(
self
.
new_messages
())
data
instance-attribute
data
:
ResultData
Data from the final response in the run.
cost
cost
()
->
Cost
Return the cost of the whole run.
Source code in
pydantic_ai_slim/pydantic_ai/result.py
def
cost
(
self
)
->
Cost
:
"""Return the cost of the whole run."""
return
self
.
_cost
StreamedRunResult
dataclass
Bases:
_BaseRunResult
[
ResultData
]
,
Generic
[
AgentDeps
,
ResultData
]
Result of a streamed run that returns structured data via a tool call.
Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
@dataclass
class StreamedRunResult(_BaseRunResult[ResultData], Generic[AgentDeps, ResultData]):
"""Result of a streamed run that returns structured data via a tool call."""
cost_so_far: Cost
"""Cost of the run up until the last request."""
_stream_response: models.EitherStreamedResponse
_result_schema: _result.ResultSchema[ResultData] | None
_deps: AgentDeps
_result_validators: list[_result
```

```
.
ResultValidator
[
AgentDeps
,
ResultData
]]
_on_complete
:
Callable
[[
list
[
messages
.
Message
]],
None
]
is_complete
:
bool
=
field
(
default
=
False
,
init
=
False
)
"""Whether the stream has all been received.
This is set to `True` when one of
[`stream`][pydantic_ai.result.StreamedRunResult.stream],
[`stream_text`][pydantic_ai.result.StreamedRunResult.stream_text],
[`stream_structured`][pydantic_ai.result.StreamedRunResult.stream_structured] or
[`get_data`][pydantic_ai.result.StreamedRunResult.get_data] completes.
"""
async
def
stream
(
self
,
*
,
debounce_by
:
float
|
None
=
0.1
)
->
AsyncIterator
[
ResultData
]:
"""Stream the response as an async iterable.
The pydantic validator for structured data will be called in
[partial mode](https://docs.pydantic.dev/dev/concepts/experimental/#partial-validation)
on each iteration.
Args:
debounce_by: by how much (if at all) to debounce/group the response chunks by. `None` means no
debouncing.
Debouncing is particularly important for long structured responses to reduce the overhead of
performing validation as each token is received.
Returns:
An async iterable of the response data.
"""
if
isinstance
(
self
```

```
.
_stream_response
,
models
.
StreamTextResponse
):
async
for
text
in
self
.
stream_text
(
debounce_by
=
debounce_by
):
yield
cast
(
ResultData
,
text
)
else
:
async
for
structured_message
,
is_last
in
self
.
stream_structured
(
debounce_by
=
debounce_by
):
yield
await
self
.
validate_structured_result
(
structured_message
,
allow_partial
=
not
is_last
)
async
def
stream_text
(
self
,
*
,
delta
:
bool
=
False
,
debounce_by
:
float
|
None
=
0.1
)
```

```
->
AsyncIterator
[
str
]:
"""Stream the text result as an async iterable.
!!! note
This method will fail if the response is structured,
e.g. if [`is_structured`][pydantic_ai.result.StreamedRunResult.is_structured] returns `True`.
!!! note
Result validators will NOT be called on the text result if `delta=True`.
Args:
delta: if `True`, yield each chunk of text as it is received, if `False` (default), yield the full
text
up to the current point.
debounce_by: by how much (if at all) to debounce/group the response chunks by. `None` means no
debouncing.
Debouncing is particularly important for long structured responses to reduce the overhead of
performing validation as each token is received.
"""
with
_logfire
.
span
(
'response stream text'
)
as
lf_span
:
if
isinstance
(
self
.
_stream_response
,
models
.
StreamStructuredResponse
):
raise
exceptions
.
UserError
(
'stream_text() can only be used with text responses'
)
if
delta
:
async
with
_utils
.
group_by_temporal
(
self
.
_stream_response
,
debounce_by
)
as
group_iter
:
async
for
_
in
group_iter
:
yield
''
.
join
(
```

```
self
.
_stream_response
.
get
())
final_delta
=
''
.
join
(
self
.
_stream_response
.
get
(
final
=
True
))
if
final_delta
:
yield
final_delta
else
:
# a quick benchmark shows it's faster to build up a string with concat when we're
# yielding at each step
chunks
:
list
[
str
]
=
[]
combined
=
''
async
with
_utils
.
group_by_temporal
(
self
.
_stream_response
,
debounce_by
)
as
group_iter
:
async
for
_
in
group_iter
:
new
=
False
for
chunk
in
self
.
_stream_response
.
get
():
chunks
.
```

```
append
(
chunk
)
new
=
True
if
new
:
combined
=
await
self
.
_validate_text_result
(
''
.
join
(
chunks
))
yield
combined
new
=
False
for
chunk
in
self
.
_stream_response
.
get
(
final
=
True
):
chunks
.
append
(
chunk
)
new
=
True
if
new
:
combined
=
await
self
.
_validate_text_result
(
''
.
join
(
chunks
))
yield
combined
lf_span
.
set_attribute
(
'combined_text'
,
combined
)
self
```

```
.
_marked_completed
(
text
=
combined
)
async
def
stream_structured
(
self
,
*
,
debounce_by
:
float
|
None
=
0.1
)
->
AsyncIterator
[
tuple
[
messages
.
ModelStructuredResponse
,
bool
]]:
"""Stream the response as an async iterable of Structured LLM Messages.
!!! note
This method will fail if the response is text,
e.g. if [`is_structured`][pydantic_ai.result.StreamedRunResult.is_structured] returns `False`.
Args:
debounce_by: by how much (if at all) to debounce/group the response chunks by. `None` means no
debouncing.
Debouncing is particularly important for long structured responses to reduce the overhead of
performing validation as each token is received.
Returns:
An async iterable of the structured response message and whether that is the last message.
"""
with
_logfire
.
span
(
'response stream structured'
)
as
lf_span
:
if
isinstance
(
self
.
_stream_response
,
models
.
StreamTextResponse
):
raise
exceptions
.
UserError
(
'stream_structured() can only be used with structured responses'
)
else
:
# we should already have a message at this point, yield that first if it has any content
```

```
msg
=
self
.
_stream_response
.
get
()
if
any
(
call
.
has_content
()
for
call
in
msg
.
calls
):
yield
msg
,
False
async
with
_utils
.
group_by_temporal
(
self
.
_stream_response
,
debounce_by
)
as
group_iter
:
async
for
_
in
group_iter
:
msg
=
self
.
_stream_response
.
get
()
if
any
(
call
.
has_content
()
for
call
in
msg
.
calls
):
yield
msg
,
False
msg
=
self
.
```

```
_stream_response
.
get
(
final
=
True
)
yield
msg
,
True
lf_span
.
set_attribute
(
'structured_response'
,
msg
)
self
.
_marked_completed
(
structured_message
=
msg
)
async
def
get_data
(
self
)
->
ResultData
:
"""Stream the whole response, validate and return it."""
async
for
_
in
self
.
_stream_response
:
pass
if
isinstance
(
self
.
_stream_response
,
models
.
StreamTextResponse
):
text
=
''
.
join
(
self
.
_stream_response
.
get
(
final
=
True
))
text
=
await
```

```
self
.
_validate_text_result
(
text
)
self
.
_marked_completed
(
text
=
text
)
return
cast
(
ResultData
,
text
)
else
:
structured_message
=
self
.
_stream_response
.
get
(
final
=
True
)
self
.
_marked_completed
(
structured_message
=
structured_message
)
return
await
self
.
validate_structured_result
(
structured_message
)
@property
def
is_structured
(
self
)
->
bool
:
"""Return whether the stream response contains structured data (as opposed to text)."""
return
isinstance
(
self
.
_stream_response
,
models
.
StreamStructuredResponse
)
def
cost
(
self
)
```

```
->
Cost
:
"""Return the cost of the whole run.
!!! note
This won't return the full cost until the stream is finished.
"""
return
self
.
cost_so_far
+
self
.
_stream_response
.
cost
()
def
timestamp
(
self
)
->
datetime
:
"""Get the timestamp of the response."""
return
self
.
_stream_response
.
timestamp
()
async
def
validate_structured_result
(
self
,
message
:
messages
.
ModelStructuredResponse
,
*
,
allow_partial
:
bool
=
False
)
->
ResultData
:
"""Validate a structured result message."""
assert
self
.
_result_schema
is
not
None
,
'Expected _result_schema to not be None'
match
=
self
.
_result_schema
.
find_tool
(
message
)
```

```
if
match
is
None
:
raise
exceptions
.
UnexpectedModelBehavior
(
f
'Invalid message, unable to find tool:
{
self
.
_result_schema
.
tool_names
()
}
'
)
call
,
result_tool
=
match
result_data
=
result_tool
.
validate
(
call
,
allow_partial
=
allow_partial
,
wrap_validation_errors
=
False
)
for
validator
in
self
.
_result_validators
:
result_data
=
await
validator
.
validate
(
result_data
,
self
.
_deps
,
0
,
call
)
return
result_data
async
def
_validate_text_result
(
self
,
text
:
```

```
str
)
->
str
:
for
validator
in
self
.
_result_validators
:
text
=
await
validator
.
validate
(
# pyright: ignore[reportAssignmentType]
text
,
# pyright: ignore[reportArgumentType]
self
.
_deps
,
0
,
None
,
)
return
text
def
_marked_completed
(
self
,
*
,
text
:
str
|
None
=
None
,
structured_message
:
messages
.
ModelStructuredResponse
|
None
=
None
)
->
None
:
self
.
is_complete
=
True
if
text
is
not
None
:
assert
structured_message
is
None
```

```
,
'Either text or structured_message should provided, not both'
self
.
_all_messages
.
append
(
messages
.
ModelTextResponse
(
content
=
text
,
timestamp
=
self
.
_stream_response
.
timestamp
())
)
else
:
assert
structured_message
is
not
None
,
'Either text or structured_message should provided, not both'
self
.
_all_messages
.
append
(
structured_message
)
self
.
_on_complete
(
self
.
_all_messages
)
```

all_messages

```
all_messages
()
->
list
[
Message
]
```

Return the history of messages.

Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
def
all_messages
(
self
)
->
list
[
messages
.
Message
]:
```

"""Return the history of messages."""
this is a method to be consistent with the other methods
return
self
.
_all_messages
all_messages_json
all_messages_json
()
->
bytes
Return all messages from
all_messages
as JSON bytes.
Source code in
pydantic_ai_slim/pydantic_ai/result.py
def
all_messages_json
(
self
)
->
bytes
:
"""Return all messages from [`all_messages`][..all_messages] as JSON bytes."""
return
messages
.
MessagesTypeAdapter
.
dump_json
(
self
.
all_messages
())
new_messages
new_messages
()
->
list
[
Message
]
Return new messages associated with this run.
System prompts and any messages from older runs are excluded.
Source code in
pydantic_ai_slim/pydantic_ai/result.py
def
new_messages
(
self
)
->
list
[
messages
.
Message
]:
"""Return new messages associated with this run.
System prompts and any messages from older runs are excluded.
"""
return
self
.
all_messages
()[

self
.
_new_message_index
:]
new_messages_json
new_messages_json
()
->
bytes
Return new messages from
new_messages
as JSON bytes.
Source code in
pydantic_ai_slim/pydantic_ai/result.py
93
94
95
def
new_messages_json
(
self
)
->
bytes
:
"""Return new messages from [`new_messages`][..new_messages] as JSON bytes."""
return
messages
.
MessagesTypeAdapter
.
dump_json
(
self
.
new_messages
())
cost_so_far
instance-attribute
cost_so_far
:
Cost
Cost of the run up until the last request.
is_complete
class-attribute
instance-attribute
is_complete
:
bool
=
field
(
default
=
False
,
init
=
False
)
Whether the stream has all been received.
This is set to
True
when one of
stream
,
stream_text
,
stream_structured
or
get_data
completes.
stream
async
stream
(
*

,
debounce_by
:
float
|
None
=
0.1
)
->
AsyncIterator
[
ResultData
]
Stream the response as an async iterable.
The pydantic validator for structured data will be called in
partial mode
on each iteration.
Parameters:
Name
Type
Description
Default
debounce_by
float
| None
by how much (if at all) to debounce/group the response chunks by.
None
means no debouncing.
Debouncing is particularly important for long structured responses to reduce the overhead of
performing validation as each token is received.
0.1
Returns:
Type
Description
AsyncIterator
[
ResultData
]
An async iterable of the response data.
Source code in
pydantic_ai_slim/pydantic_ai/result.py
async
def
stream
(
self
,
*
,
debounce_by
:
float
|
None
=

```
0.1
)
->
AsyncIterator
[
ResultData
]:
"""Stream the response as an async iterable.
The pydantic validator for structured data will be called in
[partial mode](https://docs.pydantic.dev/dev/concepts/experimental/#partial-validation)
on each iteration.
Args:
debounce_by: by how much (if at all) to debounce/group the response chunks by. `None` means no
debouncing.
Debouncing is particularly important for long structured responses to reduce the overhead of
performing validation as each token is received.
Returns:
An async iterable of the response data.
"""
if
isinstance
(
self
.
_stream_response
,
models
.
StreamTextResponse
):
async
for
text
in
self
.
stream_text
(
debounce_by
=
debounce_by
):
yield
cast
(
ResultData
,
text
)
else
:
async
for
structured_message
,
is_last
in
self
.
stream_structured
(
debounce_by
=
debounce_by
):
yield
await
self
.
validate_structured_result
(
structured_message
,
allow_partial
=
not
is_last
```

)
stream_text
async
stream_text
(
*
,
delta
:
bool
=
False
,
debounce_by
:
float
|
None
=
0.1
)
->
AsyncIterator
[
str
]
Stream the text result as an async iterable.
Note
This method will fail if the response is structured,
e.g. if
is_structured
returns
True
.
Note
Result validators will NOT be called on the text result if
delta=True
.
Parameters:
Name
Type
Description
Default
delta
bool
if
True
, yield each chunk of text as it is received, if
False
(default), yield the full text
up to the current point.
False
debounce_by
float
| None
by how much (if at all) to debounce/group the response chunks by.
None
means no debouncing.
Debouncing is particularly important for long structured responses to reduce the overhead of
performing validation as each token is received.
0.1
Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
async def stream_text(self, *, delta: bool = False, debounce_by: float | None = 0.1) -> AsyncIterator[str]:
"""Stream the text result as an async iterable.
!!! note
This method will fail if the response is structured,
e.g. if [`is_structured`][pydantic_ai.result.StreamedRunResult.is_structured] returns `True`.
!!! note
Result validators will NOT be called on the text result if `delta=True`.
Args:
delta: if `True`, yield each chunk of text as it is received, if `False` (default), yield the full
text
up to the current point.
debounce_by: by how much (if at all) to debounce/group the response chunks by. `None` means no
debouncing.
Debouncing is particularly important for long structured responses to reduce the overhead of
```

```
performing validation as each token is received.
"""
with
_logfire
.
span
(
'response stream text'
)
as
lf_span
:
if
isinstance
(
self
.
_stream_response
,
models
.
StreamStructuredResponse
):
raise
exceptions
.
UserError
(
'stream_text() can only be used with text responses'
)
if
delta
:
async
with
_utils
.
group_by_temporal
(
self
.
_stream_response
,
debounce_by
)
as
group_iter
:
async
for
_
in
group_iter
:
yield
''
.
join
(
self
.
_stream_response
.
get
())
final_delta
=
''
.
join
(
self
.
_stream_response
.
get
(
```

```
final
=
True
))
if
final_delta
:
yield
final_delta
else
:
# a quick benchmark shows it's faster to build up a string with concat when we're
# yielding at each step
chunks
:
list
[
str
]
=
[]
combined
=
''
async
with
_utils
.
group_by_temporal
(
self
.
_stream_response
,
debounce_by
)
as
group_iter
:
async
for
_
in
group_iter
:
new
=
False
for
chunk
in
self
.
_stream_response
.
get
():
chunks
.
append
(
chunk
)
new
=
True
if
new
:
combined
=
await
self
.
_validate_text_result
(
''
```

```
.
join
(
chunks
))
yield
combined
new
=
False
for
chunk
in
self
.
_stream_response
.
get
(
final
=
True
):
chunks
.
append
(
chunk
)
new
=
True
if
new
:
combined
=
await
self
.
_validate_text_result
(
''
.
join
(
chunks
))
yield
combined
lf_span
.
set_attribute
(
'combined_text'
,
combined
)
self
.
_marked_completed
(
text
=
combined
)
stream_structured
async
stream_structured
(
*
,
debounce_by
:
float
|
None
```

=
0.1
)
->
AsyncIterator
[
tuple
[
ModelStructuredResponse
,
bool
]]
Stream the response as an async iterable of Structured LLM Messages.
Note
This method will fail if the response is text,
e.g. if
is_structured
returns
False
.
Parameters:

Name	Type	Description	Default
debounce_by	float | None	by how much (if at all) to debounce/group the response chunks by. None means no debouncing. Debouncing is particularly important for long structured responses to reduce the overhead of performing validation as each token is received.	0.1

Returns:

Type	Description
AsyncIterator[tuple[ModelStructuredResponse, bool]]	An async iterable of the structured response message and whether that is the last message.

Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
async
def
stream_structured
(
self
,
*
,
debounce_by
:
float
|
None
=
0.1
)
->
AsyncIterator
[
tuple
[
messages
.
ModelStructuredResponse
,
bool
]]:
"""Stream the response as an async iterable of Structured LLM Messages.
!!! note
This method will fail if the response is text,
e.g. if [`is_structured`][pydantic_ai.result.StreamedRunResult.is_structured] returns `False`.
Args:
debounce_by: by how much (if at all) to debounce/group the response chunks by. `None` means no
debouncing.
Debouncing is particularly important for long structured responses to reduce the overhead of
performing validation as each token is received.
Returns:
An async iterable of the structured response message and whether that is the last message.
"""
with
_logfire
.
span
(
'response stream structured'
)
as
lf_span
:
if
isinstance
(
self
.
_stream_response
,
models
.
StreamTextResponse
):
raise
exceptions
.
UserError
(
'stream_structured() can only be used with structured responses'
)
else
:
# we should already have a message at this point, yield that first if it has any content
msg
=
```

```
self
.
_stream_response
.
get
()
if
any
(
call
.
has_content
()
for
call
in
msg
.
calls
):
yield
msg
,
False
async
with
_utils
.
group_by_temporal
(
self
.
_stream_response
,
debounce_by
)
as
group_iter
:
async
for
_
in
group_iter
:
msg
=
self
.
_stream_response
.
get
()
if
any
(
call
.
has_content
()
for
call
in
msg
.
calls
):
yield
msg
,
False
msg
=
self
.
_stream_response
.
```

```
get
(
final
=
True
)
yield
msg
,
True
lf_span
.
set_attribute
(
'structured_response'
,
msg
)
self
.
_marked_completed
(
structured_message
=
msg
)
```

get_data

```
async
get_data
()
->
ResultData
```

Stream the whole response, validate and return it.

Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
async
def
get_data
(
self
)
->
ResultData
:
"""Stream the whole response, validate and return it."""
async
for
_
in
self
.
_stream_response
:
pass
if
isinstance
(
self
.
_stream_response
,
models
.
StreamTextResponse
```

```
):
text
=
''
.
join
(
self
.
_stream_response
.
get
(
final
=
True
))
text
=
await
self
.
_validate_text_result
(
text
)
self
.
_marked_completed
(
text
=
text
)
return
cast
(
ResultData
,
text
)
else
:
structured_message
=
self
.
_stream_response
.
get
(
final
=
True
)
self
.
_marked_completed
(
structured_message
=
structured_message
)
return
await
self
.
validate_structured_result
(
structured_message
)
is_structured
property
is_structured
:
bool
```

Return whether the stream response contains structured data (as opposed to text).

cost
cost
()
->
Cost
Return the cost of the whole run.
Note
This won't return the full cost until the stream is finished.
Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
def
cost
(
self
)
->
Cost
:
"""Return the cost of the whole run.
!!! note
This won't return the full cost until the stream is finished.
"""
return
self
.
cost_so_far
+
self
.
_stream_response
.
cost
()
```

timestamp
timestamp
()
->
datetime
Get the timestamp of the response.
Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
def
timestamp
(
self
)
->
datetime
:
"""Get the timestamp of the response."""
return
self
.
_stream_response
.
timestamp
()
```

validate_structured_result
async
validate_structured_result
(
message
:
ModelStructuredResponse
,
*
,

```
allow_partial
:
bool
=
False
)
->
ResultData
```

Validate a structured result message.

Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
async
def
validate_structured_result
(
self
,
message
:
messages
.
ModelStructuredResponse
,
*
,
allow_partial
:
bool
=
False
)
->
ResultData
:
"""Validate a structured result message."""
assert
self
.
_result_schema
is
not
None
,
'Expected _result_schema to not be None'
match
=
self
.
_result_schema
.
find_tool
(
message
)
if
match
is
None
:
raise
```

```
exceptions
.
UnexpectedModelBehavior
(
f
'Invalid message, unable to find tool:
{
self
.
_result_schema
.
tool_names
()
}
'
)
call
,
result_tool
=
match
result_data
=
result_tool
.
validate
(
call
,
allow_partial
=
allow_partial
,
wrap_validation_errors
=
False
)
for
validator
in
self
.
_result_validators
:
result_data
=
await
validator
.
validate
(
result_data
,
self
.
_deps
,
0
,
call
)
return
result_data
```

Cost

dataclass

Cost of a request or run.

Responsibility for calculating costs is on the model used, PydanticAI simply sums the cost of requests.

You'll need to look up the documentation of the model you're using to convent "token count" costs to monetary costs.

Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
@dataclass
class Cost:
    """Cost of a request or run.
    Responsibility for calculating costs is on the model used, PydanticAI simply sums the cost of requests.
    You'll need to look up the documentation of the model you're using to convent "token count" costs to monetary costs.
    """
    request_tokens: int | None = None
    """Tokens used in processing the request."""
    response_tokens: int | None = None
    """Tokens used in generating the response."""
    total_tokens: int | None = None
    """Total tokens used in the whole run, should generally be equal to `request_tokens + response_tokens`."""
    details: dict[str, int] | None
```

```
=
None
"""Any extra details returned by the model."""
def
__add__
(
self
,
other
:
Cost
)
->
Cost
:
"""Add two costs together.
This is provided so it's trivial to sum costs from multiple requests and runs.
"""
counts
:
dict
[
str
,
int
]
=
{}
for
f
in
'request_tokens'
,
'response_tokens'
,
'total_tokens'
:
self_value
=
getattr
(
self
,
f
)
other_value
=
getattr
(
other
,
f
)
if
self_value
is
not
None
or
other_value
is
not
None
:
counts
[
f
]
=
(
self_value
or
0
)
+
(
other_value
```

```
or
0
)
details
=
self
.
details
.
copy
()
if
self
.
details
is
not
None
else
None
if
other
.
details
is
not
None
:
details
=
details
or
{}
for
key
,
value
in
other
.
details
.
items
():
details
[
key
]
=
details
.
get
(
key
,
0
)
+
value
return
Cost
(
**
counts
,
details
=
details
or
None
)
request_tokens
class-attribute
instance-attribute
request_tokens
:
int
```

|
None
=
None
Tokens used in processing the request.
response_tokens
class-attribute
instance-attribute
response_tokens
:
int
|
None
=
None
Tokens used in generating the response.
total_tokens
class-attribute
instance-attribute
total_tokens
:
int
|
None
=
None
Total tokens used in the whole run, should generally be equal to
request_tokens + response_tokens
.
details
class-attribute
instance-attribute
details
:
dict
[
str
,
int
]
|
None
=
None
Any extra details returned by the model.
__add__
__add__
(
other
:
Cost
)
->
Cost
Add two costs together.
This is provided so it's trivial to sum costs from multiple requests and runs.
Source code in
pydantic_ai_slim/pydantic_ai/result.py

```
def
__add__
(
self
,
other
:
Cost
)
->
Cost
:
"""Add two costs together.
This is provided so it's trivial to sum costs from multiple requests and runs.
"""
counts
:
dict
[
str
,
int
]
=
{}
for
f
in
'request_tokens'
,
'response_tokens'
,
'total_tokens'
:
self_value
=
getattr
(
self
,
f
)
other_value
=
getattr
(
other
,
f
)
if
self_value
is
not
None
or
other_value
is
not
None
:
counts
[
f
]
=
(
self_value
or
0
)
+
(
other_value
or
0
)
```

```
details
=
self
.
details
.
copy
()
if
self
.
details
is
not
None
else
None
if
other
.
details
is
not
None
:
details
=
details
or
{}
for
key
,
value
in
other
.
details
.
items
():
details
[
key
]
=
details
.
get
(
key
,
0
)
+
value
return
Cost
(
**
counts
,
details
=
details
or
None
)
```


==
Page: Messages and chat history - PydanticAI
URL: https://ai.pydantic.dev/message-history/
==

Messages and chat history - PydanticAI

Skip to content
PydanticAI
Messages and chat history
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

Introduction
Documentation
Messages and chat history
PydanticAI provides access to messages exchanged during an agent run. These messages can be used both to continue a coherent conversation, and to understand how an agent performed.
Accessing Messages from Results
After running an agent, you can access the messages exchanged during that run from the
result
object.
Both
RunResult
(returned by
Agent.run
,
Agent.run_sync
)
and
StreamedRunResult
(returned by
Agent.run_stream
) have the following methods:

all_messages()
: returns all messages, including messages from prior runs and system prompts. There's also a variant that returns JSON bytes,
all_messages_json()
.
new_messages()
: returns only the messages from the current run, excluding system prompts, this is generally the data you want when you want to use the messages in further runs to continue the conversation. There's also a variant that returns JSON bytes,
new_messages_json()
.
StreamedRunResult and complete messages
On
StreamedRunResult
, the messages returned from these methods will only include the final result message once the stream has finished.
E.g. you've awaited one of the following coroutines:
StreamedRunResult.stream()
StreamedRunResult.stream_text()
StreamedRunResult.stream_structured()
StreamedRunResult.get_data()
Note:
The final result message will NOT be added to result messages if you use
.stream_text(delta=True)
since in this case the result content is never built as one string.
Example of accessing methods on a
RunResult
:
run_result_messages.py

```
from
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
,
system_prompt
=
'Be a helpful assistant.'
)
result
=
agent
.
run_sync
(
'Tell me a joke.'
)
print
(
result
.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
# all messages from the run
print
(
result
.
all_messages
())
"""
[
SystemPrompt(content='Be a helpful assistant.', role='system'),
UserPrompt(
content='Tell me a joke.',
timestamp=datetime.datetime(...),
role='user',
),
ModelTextResponse(
content='Did you hear about the toothpaste scandal? They called it Colgate.',
timestamp=datetime.datetime(...),
role='model-text-response',
```

```
),
]
"""
# messages excluding system prompts
print
(
result
.
new_messages
())
"""
[
UserPrompt(
content='Tell me a joke.',
timestamp=datetime.datetime(...),
role='user',
),
ModelTextResponse(
content='Did you hear about the toothpaste scandal? They called it Colgate.',
timestamp=datetime.datetime(...),
role='model-text-response',
),
]
"""
```

(This example is complete, it can be run "as is")

Example of accessing methods on a
StreamedRunResult
:

streamed_run_result_messages.py

```
from
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
,
system_prompt
=
'Be a helpful assistant.'
)
async
def
main
():
async
with
agent
.
run_stream
(
'Tell me a joke.'
)
as
result
:
# incomplete messages before the stream finishes
print
(
result
.
all_messages
())
"""
[
SystemPrompt(content='Be a helpful assistant.', role='system'),
UserPrompt(
content='Tell me a joke.',
timestamp=datetime.datetime(...),
role='user',
),
]
"""
async
for
```

text
in
result
.
stream
():
print
(
text
)
#> Did you hear
#> Did you hear about the toothpaste
#> Did you hear about the toothpaste scandal? They called
#> Did you hear about the toothpaste scandal? They called it Colgate.
complete messages once the stream finishes
print
(
result
.
all_messages
())
"""
[
SystemPrompt(content='Be a helpful assistant.', role='system'),
UserPrompt(
content='Tell me a joke.',
timestamp=datetime.datetime(...),
role='user',
),
ModelTextResponse(
content='Did you hear about the toothpaste scandal? They called it Colgate.',
timestamp=datetime.datetime(...),
role='model-text-response',
),
]
"""
(This example is complete, it can be run "as is")
Using Messages as Input for Further Agent Runs
The primary use of message histories in PydanticAI is to maintain context across multiple agent runs.
To use existing messages in a run, pass them to the
message_history
parameter of
Agent.run
,
Agent.run_sync
or
Agent.run_stream
.
all_messages()
vs.
new_messages()
PydanticAI will inspect any messages it receives for system prompts.
If any system prompts are found in
message_history
, new system prompts are not generated,
otherwise new system prompts are generated and inserted before
message_history
in the list of messages
used in the run.
Thus you can decide whether you want to use system prompts from a previous run or generate them
again by using
all_messages()
or
new_messages()
.
Reusing messages in a conversation
from
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
,
system_prompt

```
=
'Be a helpful assistant.'
)
result1
=
agent
.
run_sync
(
'Tell me a joke.'
)
print
(
result1
.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
result2
=
agent
.
run_sync
(
'Explain?'
,
message_history
=
result1
.
new_messages
())
print
(
result2
.
data
)
#> This is an excellent joke invent by Samuel Colvin, it needs no explanation.
print
(
result2
.
all_messages
())
"""
[
SystemPrompt(content='Be a helpful assistant.', role='system'),
UserPrompt(
content='Tell me a joke.',
timestamp=datetime.datetime(...),
role='user',
),
ModelTextResponse(
content='Did you hear about the toothpaste scandal? They called it Colgate.',
timestamp=datetime.datetime(...),
role='model-text-response',
),
UserPrompt(
content='Explain?',
timestamp=datetime.datetime(...),
role='user',
),
ModelTextResponse(
content='This is an excellent joke invent by Samuel Colvin, it needs no explanation.',
timestamp=datetime.datetime(...),
role='model-text-response',
),
]
"""
```

(This example is complete, it can be run "as is")

Other ways of using messages

Since messages are defined by simple dataclasses, you can manually create and manipulate, e.g. for testing.

The message format is independent of the model used, so you can use messages in different agents, or the same agent with different models.

```
from
```

```
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
,
system_prompt
=
'Be a helpful assistant.'
)
result1
=
agent
.
run_sync
(
'Tell me a joke.'
)
print
(
result1
.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
result2
=
agent
.
run_sync
(
'Explain?'
,
model
=
'gemini-1.5-pro'
,
message_history
=
result1
.
new_messages
()
)
print
(
result2
.
data
)
#> This is an excellent joke invent by Samuel Colvin, it needs no explanation.
print
(
result2
.
all_messages
())
"""
[
SystemPrompt(content='Be a helpful assistant.', role='system'),
UserPrompt(
content='Tell me a joke.',
timestamp=datetime.datetime(...),
role='user',
),
ModelTextResponse(
content='Did you hear about the toothpaste scandal? They called it Colgate.',
timestamp=datetime.datetime(...),
role='model-text-response',
),
UserPrompt(
content='Explain?',
timestamp=datetime.datetime(...),
role='user',
```

```
),
ModelTextResponse(
content='This is an excellent joke invent by Samuel Colvin, it needs no explanation.',
timestamp=datetime.datetime(...),
role='model-text-response',
),
]
"""
```

Examples
For a more complete example of using messages in conversations, see the
chat app
example.

==
Page: Stream markdown - PydanticAI
URL: https://ai.pydantic.dev/examples/stream-markdown/
==

Stream markdown - PydanticAI

This example shows how to stream markdown from an agent, using the

rich
library to highlight the output in the terminal.
It'll run the example with both OpenAI and Google Gemini models if the required environment variables are set.
Demonstrates:
streaming text responses
Running the Example
With
dependencies installed and environment variables set
, run:
pip
uv
python
-m
pydantic_ai_examples.stream_markdown
uv
run
-m
pydantic_ai_examples.stream_markdown
Example Code
```
import
asyncio
import
os
import
logfire
from
rich.console
import
Console
,
ConsoleOptions
,
RenderResult
from
rich.live
import
Live
from
rich.markdown
import
CodeBlock
,
Markdown
from
rich.syntax
import
Syntax
from
rich.text
import
Text
from
pydantic_ai
import
Agent
from
pydantic_ai.models
import
KnownModelName
# 'if-token-present' means nothing will be sent (and the example will work) if you don't have
logfire configured
logfire
.
configure
(
send_to_logfire
=
'if-token-present'
)
agent
=
Agent
()
# models to try, and the appropriate env var
models
:
```

```
list
[
tuple
[
KnownModelName
,
str
]]
=
[
(
'gemini-1.5-flash'
,
'GEMINI_API_KEY'
),
(
'openai:gpt-4o-mini'
,
'OPENAI_API_KEY'
),
(
'groq:llama-3.1-70b-versatile'
,
'GROQ_API_KEY'
),
]
async
def
main
():
prettier_code_blocks
()
console
=
Console
()
prompt
=
'Show me a short example of using Pydantic.'
console
.
log
(
f
'Asking:
{
prompt
}
...'
,
style
=
'cyan'
)
for
model
,
env_var
in
models
:
if
env_var
in
os
.
environ
:
console
.
log
(
f
'Using model:
{
model
}
```

```
'
)
with
Live
(
''
,
console
=
console
,
vertical_overflow
=
'visible'
)
as
live
:
async
with
agent
.
run_stream
(
prompt
,
model
=
model
)
as
result
:
async
for
message
in
result
.
stream
():
live
.
update
(
Markdown
(
message
))
console
.
log
(
result
.
cost
())
else
:
console
.
log
(
f
'
{
model
}
requires
{
env_var
}
to be set.'
)
def
prettier_code_blocks
():
```

```
"""Make rich code blocks prettier and easier to copy.
From https://github.com/samuelcolvin/aicli/blob/v0.8.0/samuelcolvin_aicli.py#L22
"""
class
SimpleCodeBlock
(
CodeBlock
):
def
__rich_console__
(
self
,
console
:
Console
,
options
:
ConsoleOptions
)
->
RenderResult
:
code
=
str
(
self
.
text
)
.
rstrip
()
yield
Text
(
self
.
lexer_name
,
style
=
'dim'
)
yield
Syntax
(
code
,
self
.
lexer_name
,
theme
=
self
.
theme
,
background_color
=
'default'
,
word_wrap
=
True
,
)
yield
Text
(
f
'/
{
self
```

```
.
lexer_name
}
'
,
style
=
'dim'
)
Markdown
.
elements
[
'fence'
]
=
SimpleCodeBlock
if
__name__
==
'__main__'
:
asyncio
.
run
(
main
())
```


==
Page: Contributing - PydanticAI
URL: https://ai.pydantic.dev/contributing/
==

Contributing - PydanticAI
Skip to content
PydanticAI
Contributing
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Installation and Setup
Contributing
We'd love you to contribute to PydanticAI!
Installation and Setup
Clone your fork and cd into the repo directory
git
clone

:<your
username>/pydantic.git
cd
pydantic-ai
Install
uv
and
pre-commit
We use pipx here, for other options see:
uv
getting install docs
pre-commit
install docs
To get
pipx
itself, see
these docs
pipx
install
uv
pre-commit
Install
pydantic-ai
, deps, test deps, and docs deps
make
install

==
Page: pydantic_ai.models.vertexai - PydanticAI
URL: https://ai.pydantic.dev/api/models/vertexai/
==

pydantic_ai.models.vertexai - PydanticAI
Skip to content
PydanticAI
pydantic_ai.models.vertexai
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai
Introduction
Installation & Setup
Getting Help
Contributing
Documentation
Documentation
Agents
Dependencies
Results
Messages and chat history
Testing and Evals
Debugging and Monitoring
Examples
Examples
Pydantic Model
Weather agent
Bank support
SQL Generation

RAG
Stream markdown
Stream whales
Chat App with FastAPI
API Reference
API Reference
pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.vertexai
Table of contents
Setup
Example Usage
vertexai
VERTEX_AI_URL_TEMPLATE
VertexAIModel
__init__
BearerTokenAuth
VertexAiRegion
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Setup
Example Usage
vertexai
VERTEX_AI_URL_TEMPLATE
VertexAIModel
__init__
BearerTokenAuth
VertexAiRegion
Introduction
API Reference
pydantic_ai.models.vertexai
Custom interface to the
*-aiplatform.googleapis.com
API for Gemini models.
This model uses
GeminiAgentModel
with just the URL and auth method
changed from
GeminiModel
, it relies on the VertexAI
generateContent
and
streamGenerateContent
function endpoints
having the same schemas as the equivalent
Gemini endpoints
.
Setup
For details on how to set up authentication with this model as well as a comparison with the
generativelanguage.googleapis.com
API used by
GeminiModel
,
see
model configuration for Gemini via VertexAI
.
Example Usage
With the default google project already configured in your environment using "application default
credentials":
vertex_example_env.py
from
pydantic_ai
import
Agent
from
pydantic_ai.models.vertexai

```
import
VertexAIModel
model
=
VertexAIModel
(
'gemini-1.5-flash'
)
agent
=
Agent
(
model
)
result
=
agent
.
run_sync
(
'Tell me a joke.'
)
print
(
result
.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
```

Or using a service account JSON file:

```
vertex_example_service_account.py
from
pydantic_ai
import
Agent
from
pydantic_ai.models.vertexai
import
VertexAIModel
model
=
VertexAIModel
(
'gemini-1.5-flash'
,
service_account_file
=
'path/to/service-account.json'
,
)
agent
=
Agent
(
model
)
result
=
agent
.
run_sync
(
'Tell me a joke.'
)
print
(
result
.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
```

VERTEX_AI_URL_TEMPLATE
module-attribute

```
VERTEX_AI_URL_TEMPLATE
=
"https://
{region}
```

-aiplatform.googleapis.com/v1/projects/
{project_id}
/locations/
{region}
/publishers/
{model_publisher}
/models/
{model}
:"
URL template for Vertex AI.
See
generateContent
docs
and
streamGenerateContent
docs
for more information.
The template is used thus:
region
is substituted with the
region
argument,
see
available regions
model_publisher
is substituted with the
model_publisher
argument
model
is substituted with the
model_name
argument
project_id
is substituted with the
project_id
from auth/credentials
function
(
generateContent
or
streamGenerateContent
) is added to the end of the URL
VertexAIModel
dataclass
Bases:
Model
A model that uses Gemini via the
*-aiplatform.googleapis.com
VertexAI API.
Source code in
pydantic_ai_slim/pydantic_ai/models/vertexai.py

```
@dataclass
(
init
=
False
)
class
VertexAIModel
(
Model
):
"""A model that uses Gemini via the `*-aiplatform.googleapis.com` VertexAI API."""
model_name
:
GeminiModelName
service_account_file
:
Path
|
str
|
None
project_id
:
str
|
None
region
:
VertexAiRegion
model_publisher
:
Literal
[
'google'
]
http_client
:
AsyncHTTPClient
url_template
:
str
auth
:
BearerTokenAuth
|
None
url
:
str
|
None
# TODO __init__ can be removed once we drop 3.9 and we can set kw_only correctly on the dataclass
def
__init__
(
self
,
model_name
:
GeminiModelName
,
*
,
service_account_file
:
Path
|
str
|
```

```
None
=
None
,
project_id
:
str
|
None
=
None
,
region
:
VertexAiRegion
=
'us-central1'
,
model_publisher
:
Literal
[
'google'
]
=
'google'
,
http_client
:
AsyncHTTPClient
|
None
=
None
,
url_template
:
str
=
VERTEX_AI_URL_TEMPLATE
,
):
"""Initialize a Vertex AI Gemini model.
Args:
model_name: The name of the model to use. I couldn't find a list of supported Google models, in
VertexAI
so for now this uses the same models as the [Gemini model][pydantic_ai.models.gemini.GeminiModel].
service_account_file: Path to a service account file.
If not provided, the default environment credentials will be used.
project_id: The project ID to use, if not provided it will be taken from the credentials.
region: The region to make requests to.
model_publisher: The model publisher to use, I couldn't find a good list of available publishers,
and from trial and error it seems non-google models don't work with the `generateContent` and
`streamGenerateContent` functions, hence only `google` is currently supported.
Please create an issue or PR if you know how to use other publishers.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
url_template: URL template for Vertex AI, see
[`VERTEX_AI_URL_TEMPLATE` docs][pydantic_ai.models.vertexai.VERTEX_AI_URL_TEMPLATE]
for more information.
"""
self
.
model_name
=
model_name
self
.
service_account_file
=
service_account_file
self
.
project_id
=
project_id
self
.
```

```
region
=
region
self
.
model_publisher
=
model_publisher
self
.
http_client
=
http_client
or
cached_async_http_client
()
self
.
url_template
=
url_template
self
.
auth
=
None
self
.
url
=
None
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
],
)
->
GeminiAgentModel
:
url
,
auth
=
await
self
.
_ainit
()
return
GeminiAgentModel
(
http_client
=
self
.
http_client
,
```

```
model_name
=
self
.
model_name
,
auth
=
auth
,
url
=
url
,
function_tools
=
function_tools
,
allow_text_result
=
allow_text_result
,
result_tools
=
result_tools
,
)
async
def
_ainit
(
self
)
->
tuple
[
str
,
BearerTokenAuth
]:
if
self
.
url
is
not
None
and
self
.
auth
is
not
None
:
return
self
.
url
,
self
.
auth
if
self
.
service_account_file
is
not
None
:
creds
:
BaseCredentials
|
ServiceAccountCredentials
=
```

```
_creds_from_file
(
self
.
service_account_file
)
assert
creds
.
project_id
is
None
or
isinstance
(
creds
.
project_id
,
str
)
creds_project_id
:
str
|
None
=
creds
.
project_id
creds_source
=
'service account file'
else
:
creds
,
creds_project_id
=
await
_async_google_auth
()
creds_source
=
'`google.auth.default()`'
if
self
.
project_id
is
None
:
if
creds_project_id
is
None
:
raise
UserError
(
f
'No project_id provided and none found in
{
creds_source
}
'
)
project_id
=
creds_project_id
else
:
if
creds_project_id
is
not
None
```

```
and
self
.
project_id
!=
creds_project_id
:
raise
UserError
(
f
'The project_id you provided does not match the one from
{
creds_source
}
: '
f
'
{
self
.
project_id
!r}
!=
{
creds_project_id
!r}
'
)
project_id
=
self
.
project_id
self
.
url
=
url
=
self
.
url_template
.
format
(
region
=
self
.
region
,
project_id
=
project_id
,
model_publisher
=
self
.
model_publisher
,
model
=
self
.
model_name
,
)
self
.
auth
=
auth
=
BearerTokenAuth
(
```

```
creds
)
return
url
,
auth
def
name
(
self
)
->
str
:
return
f
'vertexai:
{
self
.
model_name
}
'
__init__
__init__
(
model_name
:
GeminiModelName
,
*
,
service_account_file
:
Path
|
str
|
None
=
None
,
project_id
:
str
|
None
=
None
,
region
:
VertexAiRegion
=
"us-central1"
,
model_publisher
:
Literal
[
"google"
]
=
"google"
,
http_client
:
AsyncClient
|
None
=
None
,
url_template
:
str
=
```

VERTEX_AI_URL_TEMPLATE
)
Initialize a Vertex AI Gemini model.
Parameters:
Name
Type
Description
Default
model_name
GeminiModelName
The name of the model to use. I couldn't find a list of supported Google models, in VertexAI so for now this uses the same models as the
Gemini model
.
required
service_account_file
Path
|
str
| None
Path to a service account file.
If not provided, the default environment credentials will be used.
None
project_id
str
| None
The project ID to use, if not provided it will be taken from the credentials.
None
region
VertexAiRegion
The region to make requests to.
'us-central1'
model_publisher
Literal
['google']
The model publisher to use, I couldn't find a good list of available publishers, and from trial and error it seems non-google models don't work with the
generateContent
and
streamGenerateContent
functions, hence only
google
is currently supported.
Please create an issue or PR if you know how to use other publishers.
'google'
http_client
AsyncClient
| None
An existing
httpx.AsyncClient
to use for making HTTP requests.
None
url_template
str
URL template for Vertex AI, see
VERTEX_AI_URL_TEMPLATE
docs
for more information.
VERTEX_AI_URL_TEMPLATE
Source code in
pydantic_ai_slim/pydantic_ai/models/vertexai.py

```
def __init__(
    self,
    model_name: GeminiModelName,
    *,
    service_account_file: Path | str | None = None,
    project_id: str | None = None,
    region: VertexAiRegion = 'us-central1',
    model_publisher: Literal['google'] = 'google',
    http_client: AsyncHTTPClient | None = None,
    url_template:
```

```
str
=
VERTEX_AI_URL_TEMPLATE
,
):
"""Initialize a Vertex AI Gemini model.
Args:
model_name: The name of the model to use. I couldn't find a list of supported Google models, in
VertexAI
so for now this uses the same models as the [Gemini model][pydantic_ai.models.gemini.GeminiModel].
service_account_file: Path to a service account file.
If not provided, the default environment credentials will be used.
project_id: The project ID to use, if not provided it will be taken from the credentials.
region: The region to make requests to.
model_publisher: The model publisher to use, I couldn't find a good list of available publishers,
and from trial and error it seems non-google models don't work with the `generateContent` and
`streamGenerateContent` functions, hence only `google` is currently supported.
Please create an issue or PR if you know how to use other publishers.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
url_template: URL template for Vertex AI, see
[`VERTEX_AI_URL_TEMPLATE` docs][pydantic_ai.models.vertexai.VERTEX_AI_URL_TEMPLATE]
for more information.
"""
self
.
model_name
=
model_name
self
.
service_account_file
=
service_account_file
self
.
project_id
=
project_id
self
.
region
=
region
self
.
model_publisher
=
model_publisher
self
.
http_client
=
http_client
or
cached_async_http_client
()
self
.
url_template
=
url_template
self
.
auth
=
None
self
.
url
=
None
```

BearerTokenAuth
dataclass
Authentication using a bearer token generated by google-auth.
Source code in
pydantic_ai_slim/pydantic_ai/models/vertexai.py
184

```python
@dataclass
class BearerTokenAuth:
    """Authentication using a bearer token generated by google-auth."""
    credentials: BaseCredentials | ServiceAccountCredentials
    token_created: datetime | None = field(default=None, init=False)

    async def headers(self) -> dict[str, str]:
        if self.credentials.token is None or self._token_expired():
            await run_in_executor(
```

```
self
.
_refresh_token
)
self
.
token_created
=
datetime
.
now
()
return
{
'Authorization'
:
f
'Bearer
{
self
.
credentials
.
token
}
'
}
def
_token_expired
(
self
)
->
bool
:
if
self
.
token_created
is
None
:
return
True
else
:
return
(
datetime
.
now
()
-
self
.
token_created
)
>
MAX_TOKEN_AGE
def
_refresh_token
(
self
)
->
str
:
self
.
credentials
.
refresh
(
Request
())
assert
isinstance
```

```
(
self
.
credentials
.
token
,
str
),
f
'Expected token to be a string, got
{
self
.
credentials
.
token
}
'
return
self
.
credentials
.
token
VertexAiRegion
module-attribute
VertexAiRegion
=
Literal
[
"us-central1"
,
"us-east1"
,
"us-east4"
,
"us-south1"
,
"us-west1"
,
"us-west2"
,
"us-west3"
,
"us-west4"
,
"us-east5"
,
"europe-central2"
,
"europe-north1"
,
"europe-southwest1"
,
"europe-west1"
,
"europe-west2"
,
"europe-west3"
,
"europe-west4"
,
"europe-west6"
,
"europe-west8"
,
"europe-west9"
,
"europe-west12"
,
"africa-south1"
,
"asia-east1"
,
"asia-east2"
,
```

"asia-northeast1"
,
"asia-northeast2"
,
"asia-northeast3"
,
"asia-south1"
,
"asia-southeast1"
,
"asia-southeast2"
,
"australia-southeast1"
,
"australia-southeast2"
,
"me-central1"
,
"me-central2"
,
"me-west1"
,
"northamerica-northeast1"
,
"northamerica-northeast2"
,
"southamerica-east1"
,
"southamerica-west1"
,
]
Regions available for Vertex AI.
More details
here
.

==
Page: pydantic_ai.tools - PydanticAI
URL: https://ai.pydantic.dev/api/tools/
==

pydantic_ai.tools - PydanticAI
Skip to content
PydanticAI
pydantic_ai.tools
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

pydantic_ai.tools

Table of contents
tools
AgentDeps
RunContext
deps
retry
tool_name
ToolParams
SystemPromptFunc
ResultValidatorFunc
ToolFuncContext
ToolFuncPlain
ToolFuncEither
ToolPrepareFunc
Tool
__init__
prepare_tool_def
run
ObjectJsonSchema
ToolDefinition
name
description
parameters_json_schema
outer_typed_dict_key

pydantic_ai.tools
AgentDeps
module-attribute
AgentDeps
=
TypeVar
(
'AgentDeps'
)
Type variable for agent dependencies.
RunContext
dataclass
Bases:
Generic

[
AgentDeps
]
Information about the current call.
Source code in
pydantic_ai_slim/pydantic_ai/tools.py

```
@dataclass
class
RunContext
(
Generic
[
AgentDeps
]):
"""Information about the current call."""
deps
:
AgentDeps
"""Dependencies for the agent."""
retry
:
int
"""Number of retries so far."""
tool_name
:
str
|
None
=
None
"""Name of the tool being called."""
```

deps
instance-attribute
deps
:
AgentDeps
Dependencies for the agent.
retry
instance-attribute
retry
:
int
Number of retries so far.
tool_name
class-attribute
instance-attribute
tool_name
:
str
|
None
=
None
Name of the tool being called.
ToolParams
module-attribute
ToolParams
=
ParamSpec
(
'ToolParams'
)
Retrieval function param spec.
SystemPromptFunc
module-attribute
SystemPromptFunc
=

Union
[
Callable
[[
RunContext
[
AgentDeps
]],
str
],
Callable
[[
RunContext
[
AgentDeps
]],
Awaitable
[
str
]],
Callable
[[],
str
],
Callable
[[],
Awaitable
[
str
]],
]
A function that may or maybe not take
RunContext
as an argument, and may or may not be async.
Usage
SystemPromptFunc[AgentDeps]
.
ResultValidatorFunc
module-attribute
ResultValidatorFunc
=
Union
[
Callable
[
[
RunContext
[
AgentDeps
],
ResultData
],
ResultData
],
Callable
[
[
RunContext
[
AgentDeps
],
ResultData
],
Awaitable
[
ResultData
],
],
Callable
[[
ResultData
],
ResultData
],
Callable
[[
ResultData

],
Awaitable
[
ResultData
]],
]
A function that always takes
ResultData
and returns
ResultData
,
but may or maybe not take
CallInfo
as a first argument, and may or may not be async.
Usage
ResultValidator[AgentDeps, ResultData]
.
ToolFuncContext
module-attribute
ToolFuncContext
=
Callable
[
Concatenate
[
RunContext
[
AgentDeps
],
ToolParams
],
Any
]
A tool function that takes
RunContext
as the first argument.
Usage
ToolContextFunc[AgentDeps, ToolParams]
.
ToolFuncPlain
module-attribute
ToolFuncPlain
=
Callable
[
ToolParams
,
Any
]
A tool function that does not take
RunContext
as the first argument.
Usage
ToolPlainFunc[ToolParams]
.
ToolFuncEither
module-attribute
ToolFuncEither
=
Union
[
ToolFuncContext
[
AgentDeps
,
ToolParams
],
ToolFuncPlain
[
ToolParams
],
]
Either kind of tool function.
This is just a union of
ToolFuncContext
and
ToolFuncPlain

.
Usage
ToolFuncEither[AgentDeps, ToolParams]
.
ToolPrepareFunc
module-attribute
ToolPrepareFunc
:
TypeAlias
=
(
"Callable[[RunContext[AgentDeps], ToolDefinition], Awaitable[ToolDefinition | None]]"
)
Definition of a function that can prepare a tool definition at call time.
See
tool docs
for more information.
Example — here
only_if_42
is valid as a
ToolPrepareFunc
:
```
from
typing
import
Union
from
pydantic_ai
import
RunContext
,
Tool
from
pydantic_ai.tools
import
ToolDefinition
async
def
only_if_42
(
ctx
:
RunContext
[
int
],
tool_def
:
ToolDefinition
)
->
Union
[
ToolDefinition
,
None
]:
if
ctx
.
deps
==
42
:
return
tool_def
def
hitchhiker
(
ctx
:
RunContext
[
int
],
answer
:
```

str
)
->
str
:
return
f
'
{
ctx
.
deps
}
{
answer
}
'
hitchhiker
=
Tool
(
hitchhiker
,
prepare
=
only_if_42
)
Usage
ToolPrepareFunc[AgentDeps]
.
Tool
dataclass
Bases:
Generic
[
AgentDeps
]
A tool function for an agent.
Source code in
pydantic_ai_slim/pydantic_ai/tools.py

```python
@dataclass(init=False)
class Tool(Generic[AgentDeps]):
    """A tool function for an agent."""
    function: ToolFuncEither[AgentDeps, ...]
    takes_ctx: bool
```

```
max_retries
:
int
|
None
name
:
str
description
:
str
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
_is_async
:
bool
=
field
(
init
=
False
)
_single_arg_name
:
str
|
None
=
field
(
init
=
False
)
_positional_fields
:
list
[
str
]
=
field
(
init
=
False
)
_var_positional_field
:
str
|
None
=
field
(
init
=
False
)
_validator
:
SchemaValidator
=
field
(
init
=
False
,
repr
```

```
=
False
)
_parameters_json_schema
:
ObjectJsonSchema
=
field
(
init
=
False
)
current_retry
:
int
=
field
(
default
=
0
,
init
=
False
)
def
__init__
(
self
,
function
:
ToolFuncEither
[
AgentDeps
,
...
],
*
,
takes_ctx
:
bool
|
None
=
None
,
max_retries
:
int
|
None
=
None
,
name
:
str
|
None
=
None
,
description
:
str
|
None
=
None
,
prepare
:
ToolPrepareFunc
```

```
[
AgentDeps
]
|
None
=
None
,
):
"""Create a new tool instance.
Example usage:
```python
from pydantic_ai import Agent, RunContext, Tool
async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
return f'{ctx.deps} {x} {y}'
agent = Agent('test', tools=[Tool(my_tool)])
```
or with a custom prepare method:
```python
from typing import Union
from pydantic_ai import Agent, RunContext, Tool
from pydantic_ai.tools import ToolDefinition
async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
return f'{ctx.deps} {x} {y}'
async def prep_my_tool(
ctx: RunContext[int], tool_def: ToolDefinition
) -> Union[ToolDefinition, None]:
# only register the tool if `deps == 42`
if ctx.deps == 42:
return tool_def
agent = Agent('test', tools=[Tool(my_tool, prepare=prep_my_tool)])
```
Args:
function: The Python function to call as the tool.
takes_ctx: Whether the function takes a [`RunContext`][pydantic_ai.tools.RunContext] first argument,
this is inferred if unset.
max_retries: Maximum number of retries allowed for this tool, set to the agent default if `None`.
name: Name of the tool, inferred from the function if `None`.
description: Description of the tool, inferred from the function if `None`.
prepare: custom method to prepare the tool definition for each step, return `None` to omit this
tool from a given step. This is useful if you want to customise a tool at call time,
or omit it completely from a step. See [`ToolPrepareFunc`][pydantic_ai.tools.ToolPrepareFunc].
"""
if
takes_ctx
is
None
:
takes_ctx
=
_pydantic
.
takes_ctx
(
function
)
f
=
_pydantic
.
function_schema
(
function
,
takes_ctx
)
self
.
function
=
function
self
.
takes_ctx
=
takes_ctx
self
```

```
.
max_retries
=
max_retries
self
.
name
=
name
or
function
.
__name__
self
.
description
=
description
or
f
[
'description'
]
self
.
prepare
=
prepare
self
.
_is_async
=
inspect
.
iscoroutinefunction
(
self
.
function
)
self
.
_single_arg_name
=
f
[
'single_arg_name'
]
self
.
_positional_fields
=
f
[
'positional_fields'
]
self
.
_var_positional_field
=
f
[
'var_positional_field'
]
self
.
_validator
=
f
[
'validator'
]
self
.
_parameters_json_schema
=
f
```

```
[
'json_schema'
]
async
def
prepare_tool_def
(
self
,
ctx
:
RunContext
[
AgentDeps
])
->
ToolDefinition
|
None
:
"""Get the tool definition.
By default, this method creates a tool definition, then either returns it, or calls `self.prepare`
if it's set.
Returns:
return a `ToolDefinition` or `None` if the tools should not be registered for this run.
"""
tool_def
=
ToolDefinition
(
name
=
self
.
name
,
description
=
self
.
description
,
parameters_json_schema
=
self
.
_parameters_json_schema
,
)
if
self
.
prepare
is
not
None
:
return
await
self
.
prepare
(
ctx
,
tool_def
)
else
:
return
tool_def
async
def
run
(
self
,
```

```
deps
:
AgentDeps
,
message
:
messages
.
ToolCall
)
->
messages
.
Message
:
"""Run the tool function asynchronously."""
try
:
if
isinstance
(
message
.
args
,
messages
.
ArgsJson
):
args_dict
=
self
.
_validator
.
validate_json
(
message
.
args
.
args_json
)
else
:
args_dict
=
self
.
_validator
.
validate_python
(
message
.
args
.
args_dict
)
except
ValidationError
as
e
:
return
self
.
_on_error
(
e
,
message
)
args
,
kwargs
=
```

```
self
.
_call_args
(
deps
,
args_dict
,
message
)
try
:
if
self
.
_is_async
:
function
=
cast
(
Callable
[[
Any
],
Awaitable
[
str
]],
self
.
function
)
response_content
=
await
function
(
*
args
,
**
kwargs
)
else
:
function
=
cast
(
Callable
[[
Any
],
str
],
self
.
function
)
response_content
=
await
_utils
.
run_in_executor
(
function
,
*
args
,
**
kwargs
)
except
ModelRetry
```

```
as
e
:
return
self
.
_on_error
(
e
,
message
)
self
.
current_retry
=
0
return
messages
.
ToolReturn
(
tool_name
=
message
.
tool_name
,
content
=
response_content
,
tool_call_id
=
message
.
tool_call_id
,
)
def
_call_args
(
self
,
deps
:
AgentDeps
,
args_dict
:
dict
[
str
,
Any
],
message
:
messages
.
ToolCall
)
->
tuple
[
list
[
Any
],
dict
[
str
,
Any
]]:
if
self
```

```
.
_single_arg_name
:
args_dict
=
{
self
.
_single_arg_name
:
args_dict
}
args
=
[
RunContext
(
deps
,
self
.
current_retry
,
message
.
tool_name
)]
if
self
.
takes_ctx
else
[]
for
positional_field
in
self
.
_positional_fields
:
args
.
append
(
args_dict
.
pop
(
positional_field
))
if
self
.
_var_positional_field
:
args
.
extend
(
args_dict
.
pop
(
self
.
_var_positional_field
))
return
args
,
args_dict
def
_on_error
(
self
,
exc
```

```
:
ValidationError
|
ModelRetry
,
call_message
:
messages
.
ToolCall
)
->
messages
.
RetryPrompt
:
self
.
current_retry
+=
1
if
self
.
max_retries
is
None
or
self
.
current_retry
>
self
.
max_retries
:
raise
UnexpectedModelBehavior
(
f
'Tool exceeded max retries count of
{
self
.
max_retries
}
'
)
from
exc
else
:
if
isinstance
(
exc
,
ValidationError
):
content
=
exc
.
errors
(
include_url
=
False
)
else
:
content
=
exc
.
message
return
```

```
messages
.
RetryPrompt
(
tool_name
=
call_message
.
tool_name
,
content
=
content
,
tool_call_id
=
call_message
.
tool_call_id
,
)
__init__
__init__
(
function
:
ToolFuncEither
[
AgentDeps
,
...
],
*
,
takes_ctx
:
bool
|
None
=
None
,
max_retries
:
int
|
None
=
None
,
name
:
str
|
None
=
None
,
description
:
str
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
)
```

Create a new tool instance.
Example usage:
```
from
pydantic_ai
import
Agent
,
RunContext
,
Tool
async
def
my_tool
(
ctx
:
RunContext
[
int
],
x
:
int
,
y
:
int
)
->
str
:
return
f
'
{
ctx
.
deps
}
{
x
}
{
y
}
'
agent
=
Agent
(
'test'
,
tools
=
[
Tool
(
my_tool
)])
```
or with a custom prepare method:
```
from
typing
import
Union
from
pydantic_ai
import
Agent
,
RunContext
,
Tool
from
pydantic_ai.tools
import
ToolDefinition
async
```

```
def
my_tool
(
ctx
:
RunContext
[
int
],
x
:
int
,
y
:
int
)
->
str
:
return
f
'
{
ctx
.
deps
}
{
x
}
{
y
}
'
async
def
prep_my_tool
(
ctx
:
RunContext
[
int
],
tool_def
:
ToolDefinition
)
->
Union
[
ToolDefinition
,
None
]:
# only register the tool if `deps == 42`
if
ctx
.
deps
==
42
:
return
tool_def
agent
=
Agent
(
'test'
,
tools
=
[
Tool
(
```

my_tool
,
prepare
=
prep_my_tool
)])
Parameters:
Name
Type
Description
Default
function
ToolFuncEither
[
AgentDeps
, ...]
The Python function to call as the tool.
required
takes_ctx
bool
| None
Whether the function takes a
RunContext
first argument,
this is inferred if unset.
None
max_retries
int
| None
Maximum number of retries allowed for this tool, set to the agent default if
None
.
None
name
str
| None
Name of the tool, inferred from the function if
None
.
None
description
str
| None
Description of the tool, inferred from the function if
None
.
None
prepare
ToolPrepareFunc
[
AgentDeps
] | None
custom method to prepare the tool definition for each step, return
None
to omit this
tool from a given step. This is useful if you want to customise a tool at call time,
or omit it completely from a step. See
ToolPrepareFunc
.
None
Source code in
pydantic_ai_slim/pydantic_ai/tools.py

```python
def __init__(self, function: ToolFuncEither[AgentDeps, ...], *, takes_ctx: bool | None
```

```
=
None
,
max_retries
:
int
|
None
=
None
,
name
:
str
|
None
=
None
,
description
:
str
|
None
=
None
,
prepare
:
ToolPrepareFunc
[
AgentDeps
]
|
None
=
None
,
):
"""Create a new tool instance.
Example usage:
```python
from pydantic_ai import Agent, RunContext, Tool
async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
return f'{ctx.deps} {x} {y}'
agent = Agent('test', tools=[Tool(my_tool)])
```
or with a custom prepare method:
```python
from typing import Union
from pydantic_ai import Agent, RunContext, Tool
from pydantic_ai.tools import ToolDefinition
async def my_tool(ctx: RunContext[int], x: int, y: int) -> str:
return f'{ctx.deps} {x} {y}'
async def prep_my_tool(
ctx: RunContext[int], tool_def: ToolDefinition
) -> Union[ToolDefinition, None]:
# only register the tool if `deps == 42`
if ctx.deps == 42:
return tool_def
agent = Agent('test', tools=[Tool(my_tool, prepare=prep_my_tool)])
```
Args:
function: The Python function to call as the tool.
takes_ctx: Whether the function takes a [`RunContext`][pydantic_ai.tools.RunContext] first argument,
this is inferred if unset.
max_retries: Maximum number of retries allowed for this tool, set to the agent default if `None`.
name: Name of the tool, inferred from the function if `None`.
description: Description of the tool, inferred from the function if `None`.
prepare: custom method to prepare the tool definition for each step, return `None` to omit this
tool from a given step. This is useful if you want to customise a tool at call time,
or omit it completely from a step. See [`ToolPrepareFunc`][pydantic_ai.tools.ToolPrepareFunc].
"""
if
takes_ctx
is
None
```

```
:
takes_ctx
=
_pydantic
.
takes_ctx
(
function
)
f
=
_pydantic
.
function_schema
(
function
,
takes_ctx
)
self
.
function
=
function
self
.
takes_ctx
=
takes_ctx
self
.
max_retries
=
max_retries
self
.
name
=
name
or
function
.
__name__
self
.
description
=
description
or
f
[
'description'
]
self
.
prepare
=
prepare
self
.
_is_async
=
inspect
.
iscoroutinefunction
(
self
.
function
)
self
.
_single_arg_name
=
f
[
'single_arg_name'
```

]
self
.
_positional_fields
=
f
[
'positional_fields'
]
self
.
_var_positional_field
=
f
[
'var_positional_field'
]
self
.
_validator
=
f
[
'validator'
]
self
.
_parameters_json_schema
=
f
[
'json_schema'
]
prepare_tool_def
async
prepare_tool_def
(
ctx
:
RunContext
[
AgentDeps
],
)
->
ToolDefinition
|
None
Get the tool definition.
By default, this method creates a tool definition, then either returns it, or calls
self.prepare
if it's set.
Returns:
Type
Description
ToolDefinition
| None
return a
ToolDefinition
or
None
if the tools should not be registered for this run.
Source code in
pydantic_ai_slim/pydantic_ai/tools.py

```
async def prepare_tool_def(self, ctx: RunContext[AgentDeps]) -> ToolDefinition | None:
    """Get the tool definition.
    By default, this method creates a tool definition, then either returns it, or calls `self.prepare`
    if it's set.
    Returns:
    return a `ToolDefinition` or `None` if the tools should not be registered for this run.
    """
    tool_def = ToolDefinition(
        name=self.name,
        description=self.description,
        parameters_json_schema=self._parameters_json_schema,
    )
    if self.prepare is not None:
        return await self.prepare(ctx, tool_def)
    else:
        return tool_def
run
async run(
```

```
deps
:
AgentDeps
,
message
:
ToolCall
)
->
Message
```

Run the tool function asynchronously.

Source code in
pydantic_ai_slim/pydantic_ai/tools.py

```
async
def
run
(
self
,
deps
:
AgentDeps
,
message
:
messages
.
ToolCall
)
->
messages
.
Message
:
"""Run the tool function asynchronously."""
try
:
if
isinstance
(
message
.
args
,
messages
.
ArgsJson
):
args_dict
=
```

```
self
.
_validator
.
validate_json
(
message
.
args
.
args_json
)
else
:
args_dict
=
self
.
_validator
.
validate_python
(
message
.
args
.
args_dict
)
except
ValidationError
as
e
:
return
self
.
_on_error
(
e
,
message
)
args
,
kwargs
=
self
.
_call_args
(
deps
,
args_dict
,
message
)
try
:
if
self
.
_is_async
:
function
=
cast
(
Callable
[[
Any
],
Awaitable
[
str
]],
self
.
```

```
function
)
response_content
=
await
function
(
*
args
,
**
kwargs
)
else
:
function
=
cast
(
Callable
[[
Any
],
str
],
self
.
function
)
response_content
=
await
_utils
.
run_in_executor
(
function
,
*
args
,
**
kwargs
)
except
ModelRetry
as
e
:
return
self
.
_on_error
(
e
,
message
)
self
.
current_retry
=
0
return
messages
.
ToolReturn
(
tool_name
=
message
.
tool_name
,
content
=
response_content
```

,
tool_call_id
=
message
.
tool_call_id
,
)
ObjectJsonSchema
module-attribute
ObjectJsonSchema
:
TypeAlias
=
dict
[
str
,
Any
]
Type representing JSON schema of an object, e.g. where
"type": "object"
.
This type is used to define tools parameters (aka arguments) in
ToolDefinition
.
With PEP-728 this should be a TypedDict with
type: Literal['object']
, and
extra_items=Any
ToolDefinition
dataclass
Definition of a tool passed to a model.
This is used for both function tools result tools.
Source code in
pydantic_ai_slim/pydantic_ai/tools.py

```
@dataclass
class
ToolDefinition
:
"""Definition of a tool passed to a model.
This is used for both function tools result tools.
"""
name
:
str
"""The name of the tool."""
description
:
str
"""The description of the tool."""
parameters_json_schema
:
ObjectJsonSchema
"""The JSON schema for the tool's parameters."""
outer_typed_dict_key
```

:
str
|
None
=
None
"""The key in the outer [TypedDict] that wraps a result tool.
This will only be set for result tools which don't have an `object` JSON schema.
"""
name
instance-attribute
name
:
str
The name of the tool.
description
instance-attribute
description
:
str
The description of the tool.
parameters_json_schema
instance-attribute
parameters_json_schema
:
ObjectJsonSchema
The JSON schema for the tool's parameters.
outer_typed_dict_key
class-attribute
instance-attribute
outer_typed_dict_key
:
str
|
None
=
None
The key in the outer [TypedDict] that wraps a result tool.
This will only be set for result tools which don't have an
object
JSON schema.

==
Page: pydantic_ai.models.groq - PydanticAI
URL: https://ai.pydantic.dev/api/models/groq/
==

pydantic_ai.models.groq - PydanticAI

Stream whales
Chat App with FastAPI
API Reference
API Reference
pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.groq
Table of contents
Setup
groq
GroqModelName
GroqModel
__init__
GroqAgentModel
GroqStreamTextResponse
GroqStreamStructuredResponse
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Setup
groq
GroqModelName
GroqModel
__init__
GroqAgentModel
GroqStreamTextResponse
GroqStreamStructuredResponse
Introduction
API Reference
pydantic_ai.models.groq
Setup
For details on how to set up authentication with this model, see
model configuration for Groq
.
GroqModelName
module-attribute
GroqModelName
=
Literal
[
"llama-3.3-70b-versatile"
,
"llama-3.1-70b-versatile"
,
"llama3-groq-70b-8192-tool-use-preview"
,
"llama3-groq-8b-8192-tool-use-preview"
,
"llama-3.1-70b-specdec"
,
"llama-3.1-8b-instant"
,
"llama-3.2-1b-preview"
,
"llama-3.2-3b-preview"
,
"llama-3.2-11b-vision-preview"
,
"llama-3.2-90b-vision-preview"
,
"llama3-70b-8192"
,
"llama3-8b-8192"
,
"mixtral-8x7b-32768"
,
"gemma2-9b-it"

,
"gemma-7b-it"
,
]
Named Groq models.
See
the Groq docs
for a full list.
GroqModel
dataclass
Bases:
Model
A model that uses the Groq API.
Internally, this uses the
Groq Python client
to interact with the API.
Apart from
__init__
, all methods are private or match those of the base class.
Source code in
pydantic_ai_slim/pydantic_ai/models/groq.py

```python
@dataclass(init=False)
class GroqModel(Model):
    """A model that uses the Groq API.
    Internally, this uses the [Groq Python client](https://github.com/groq/groq-python) to interact with the API.
    Apart from `__init__`, all methods are private or match those of the base class.
    """
    model_name: GroqModelName
    client: AsyncGroq = field(repr=False)
    def __init__(
        self,
        model_name: GroqModelName,
        *,
        api_key: str | None = None,
        groq_client: AsyncGroq | None = None,
        http_client: AsyncHTTPClient |
```

```
None
=
None
,
):
"""Initialize a Groq model.
Args:
model_name: The name of the Groq model to use. List of model names available
[here](https://console.groq.com/docs/models).
api_key: The API key to use for authentication, if not provided, the `GROQ_API_KEY` environment
variable
will be used if available.
groq_client: An existing
[`AsyncGroq`](https://github.com/groq/groq-python?tab=readme-ov-file#async-usage)
client to use, if provided, `api_key` and `http_client` must be `None`.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
"""
self
.
model_name
=
model_name
if
groq_client
is
not
None
:
assert
http_client
is
None
,
'Cannot provide both `groq_client` and `http_client`'
assert
api_key
is
None
,
'Cannot provide both `groq_client` and `api_key`'
self
.
client
=
groq_client
elif
http_client
is
not
None
:
self
.
client
=
AsyncGroq
(
api_key
=
api_key
,
http_client
=
http_client
)
else
:
self
.
client
=
AsyncGroq
(
api_key
=
api_key
,
```

```
http_client
=
cached_async_http_client
())
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
],
)
->
AgentModel
:
check_allow_model_requests
()
tools
=
[
self
.
_map_tool_definition
(
r
)
for
r
in
function_tools
]
if
result_tools
:
tools
+=
[
self
.
_map_tool_definition
(
r
)
for
r
in
result_tools
]
return
GroqAgentModel
(
self
.
client
,
self
.
model_name
,
allow_text_result
```

```
,
tools
,
)
def
name
(
self
)
->
str
:
return
f
'groq:
{
self
.
model_name
}
'
@staticmethod
def
_map_tool_definition
(
f
:
ToolDefinition
)
->
chat
.
ChatCompletionToolParam
:
return
{
'type'
:
'function'
,
'function'
:
{
'name'
:
f
.
name
,
'description'
:
f
.
description
,
'parameters'
:
f
.
parameters_json_schema
,
},
}
__init__
__init__
(
model_name
:
GroqModelName
,
*
,
api_key
:
str
|
None
```

=
None
,
groq_client
:
AsyncGroq
|
None
=
None
,
http_client
:
AsyncClient
|
None
=
None
)
Initialize a Groq model.
Parameters:
Name
Type
Description
Default
model_name
GroqModelName
The name of the Groq model to use. List of model names available
here
.
required
api_key
str
| None
The API key to use for authentication, if not provided, the
GROQ_API_KEY
environment variable
will be used if available.
None
groq_client
AsyncGroq
| None
An existing
AsyncGroq
client to use, if provided,
api_key
and
http_client
must be
None
.
None
http_client
AsyncClient
| None
An existing
httpx.AsyncClient
to use for making HTTP requests.
None
Source code in
pydantic_ai_slim/pydantic_ai/models/groq.py

```
def __init__(
    self,
    model_name: GroqModelName,
    *,
    api_key: str | None = None,
    groq_client: AsyncGroq | None = None,
    http_client: AsyncHTTPClient | None = None,
):
    """Initialize a Groq model.
    Args:
    model_name: The name of the Groq model to use. List of model names available
    [here](https://console.groq.com/docs/models).
    api_key: The API key to use for authentication, if not provided, the `GROQ_API_KEY` environment
    variable
    will be used if available.
    groq_client: An existing
    [`AsyncGroq`](https://github.com/groq/groq-python?tab=readme-ov-file#async-usage)
    client to use, if provided, `api_key` and `http_client` must be `None`.
    http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
    """
    self.model_name = model_name
    if groq_client is not None:
        assert http_client is None,
```

```
'Cannot provide both `groq_client` and `http_client`'
assert
api_key
is
None
,
'Cannot provide both `groq_client` and `api_key`'
self
.
client
=
groq_client
elif
http_client
is
not
None
:
self
.
client
=
AsyncGroq
(
api_key
=
api_key
,
http_client
=
http_client
)
else
:
self
.
client
=
AsyncGroq
(
api_key
=
api_key
,
http_client
=
cached_async_http_client
())
```

GroqAgentModel
dataclass
Bases:
AgentModel
Implementation of
AgentModel
for Groq models.
Source code in
pydantic_ai_slim/pydantic_ai/models/groq.py

```python
@dataclass
class GroqAgentModel(AgentModel):
    """Implementation of `AgentModel` for Groq models."""

    client: AsyncGroq
    model_name: str
    allow_text_result: bool
    tools: list[chat.ChatCompletionToolParam]

    async def request(self, messages: list[Message]) -> tuple[ModelAnyResponse, result.Cost
```

```
]:
response
=
await
self
.
_completions_create
(
messages
,
False
)
return
self
.
_process_response
(
response
),
_map_cost
(
response
)
@asynccontextmanager
async
def
request_stream
(
self
,
messages
:
list
[
Message
])
->
AsyncIterator
[
EitherStreamedResponse
]:
response
=
await
self
.
_completions_create
(
messages
,
True
)
async
with
response
:
yield
await
self
.
_process_streamed_response
(
response
)
@overload
async
def
_completions_create
(
self
,
messages
:
list
[
Message
],
```

```
stream
:
Literal
[
True
]
)
->
AsyncStream
[
ChatCompletionChunk
]:
pass
@overload
async
def
_completions_create
(
self
,
messages
:
list
[
Message
],
stream
:
Literal
[
False
])
->
chat
.
ChatCompletion
:
pass
async
def
_completions_create
(
self
,
messages
:
list
[
Message
],
stream
:
bool
)
->
chat
.
ChatCompletion
|
AsyncStream
[
ChatCompletionChunk
]:
# standalone function to make it easier to override
if
not
self
.
tools
:
tool_choice
:
Literal
[
'none'
,
'required'
```

```
,
'auto'
]
|
None
=
None
elif
not
self
.
allow_text_result
:
tool_choice
=
'required'
else
:
tool_choice
=
'auto'
groq_messages
=
[
self
.
_map_message
(
m
)
for
m
in
messages
]
return
await
self
.
client
.
chat
.
completions
.
create
(
model
=
str
(
self
.
model_name
),
messages
=
groq_messages
,
temperature
=
0.0
,
n
=
1
,
parallel_tool_calls
=
True
if
self
.
tools
else
NOT_GIVEN
,
```

```
tools
=
self
.
tools
or
NOT_GIVEN
,
tool_choice
=
tool_choice
or
NOT_GIVEN
,
stream
=
stream
,
)
@staticmethod
def
_process_response
(
response
:
chat
.
ChatCompletion
)
->
ModelAnyResponse
:
"""Process a non-streamed response, and prepare a message to return."""
timestamp
=
datetime
.
fromtimestamp
(
response
.
created
,
tz
=
timezone
.
utc
)
choice
=
response
.
choices
[
0
]
if
choice
.
message
.
tool_calls
is
not
None
:
return
ModelStructuredResponse
(
[
ToolCall
.
from_json
(
c
.
```

```
function
.
name
,
c
.
function
.
arguments
,
c
.
id
)
for
c
in
choice
.
message
.
tool_calls
],
timestamp
=
timestamp
,
)
else
:
assert
choice
.
message
.
content
is
not
None
,
choice
return
ModelTextResponse
(
choice
.
message
.
content
,
timestamp
=
timestamp
)
@staticmethod
async
def
_process_streamed_response
(
response
:
AsyncStream
[
ChatCompletionChunk
])
->
EitherStreamedResponse
:
"""Process a streamed response, and prepare a streaming response to return."""
timestamp
:
datetime
|
None
=
None
start_cost
```

```
=
Cost
()
# the first chunk may contain enough information so we iterate until we get either `tool_calls` or
`content`
while
True
:
try
:
chunk
=
await
response
.
__anext__
()
except
StopAsyncIteration
as
e
:
raise
UnexpectedModelBehavior
(
'Streamed response ended without content or tool calls'
)
from
e
timestamp
=
timestamp
or
datetime
.
fromtimestamp
(
chunk
.
created
,
tz
=
timezone
.
utc
)
start_cost
+=
_map_cost
(
chunk
)
if
chunk
.
choices
:
delta
=
chunk
.
choices
[
0
]
.
delta
if
delta
.
content
is
not
None
:
return
```

```
GroqStreamTextResponse
(
delta
.
content
,
response
,
timestamp
,
start_cost
)
elif
delta
.
tool_calls
is
not
None
:
return
GroqStreamStructuredResponse
(
response
,
{
c
.
index
:
c
for
c
in
delta
.
tool_calls
},
timestamp
,
start_cost
,
)
@staticmethod
def
_map_message
(
message
:
Message
)
->
chat
.
ChatCompletionMessageParam
:
"""Just maps a `pydantic_ai.Message` to a `groq.types.ChatCompletionMessageParam`."""
if
message
.
role
==
'system'
:
# SystemPrompt ->
return
chat
.
ChatCompletionSystemMessageParam
(
role
=
'system'
,
content
=
message
```

```
.
content
)
elif
message
.
role
==
'user'
:
# UserPrompt ->
return
chat
.
ChatCompletionUserMessageParam
(
role
=
'user'
,
content
=
message
.
content
)
elif
message
.
role
==
'tool-return'
:
# ToolReturn ->
return
chat
.
ChatCompletionToolMessageParam
(
role
=
'tool'
,
tool_call_id
=
_guard_tool_call_id
(
t
=
message
,
model_source
=
'Groq'
),
content
=
message
.
model_response_str
(),
)
elif
message
.
role
==
'retry-prompt'
:
# RetryPrompt ->
if
message
.
tool_name
is
None
:
```

```
return
chat
.
ChatCompletionUserMessageParam
(
role
=
'user'
,
content
=
message
.
model_response
())
else
:
return
chat
.
ChatCompletionToolMessageParam
(
role
=
'tool'
,
tool_call_id
=
_guard_tool_call_id
(
t
=
message
,
model_source
=
'Groq'
),
content
=
message
.
model_response
(),
)
elif
message
.
role
==
'model-text-response'
:
# ModelTextResponse ->
return
chat
.
ChatCompletionAssistantMessageParam
(
role
=
'assistant'
,
content
=
message
.
content
)
elif
message
.
role
==
'model-structured-response'
:
# ModelStructuredResponse ->
return
```

```
chat
.
ChatCompletionAssistantMessageParam
(
role
=
'assistant'
,
tool_calls
=
[
_map_tool_call
(
t
)
for
t
in
message
.
calls
],
)
else
:
assert_never
(
message
)
```

GroqStreamTextResponse
dataclass
Bases:
StreamTextResponse
Implementation of
StreamTextResponse
for Groq models.
Source code in
pydantic_ai_slim/pydantic_ai/models/groq.py

```
@dataclass
class
GroqStreamTextResponse
(
StreamTextResponse
):
"""Implementation of `StreamTextResponse` for Groq models."""
_first
:
str
|
None
_response
:
AsyncStream
[
ChatCompletionChunk
]
_timestamp
:
datetime
_cost
:
result
.
Cost
_buffer
:
list
[
str
]
=
field
(
default_factory
=
list
,
init
=
False
)
async
def
__anext__
(
self
)
->
None
:
if
self
.
_first
is
not
None
:
self
.
_buffer
.
append
(
self
.
_first
)
self
.
_first
=
None
return
None
```

```
chunk
=
await
self
.
_response
.
__anext__
()
self
.
_cost
=
_map_cost
(
chunk
)
try
:
choice
=
chunk
.
choices
[
0
]
except
IndexError
:
raise
StopAsyncIteration
()
# we don't raise StopAsyncIteration on the last chunk because usage comes after this
if
choice
.
finish_reason
is
None
:
assert
choice
.
delta
.
content
is
not
None
,
f
'Expected delta with content, invalid chunk:
{
chunk
!r}
'
if
choice
.
delta
.
content
is
not
None
:
self
.
_buffer
.
append
(
choice
.
delta
.
```

```
content
)
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
Iterable
[
str
]:
yield from
self
.
_buffer
self
.
_buffer
.
clear
()
def
cost
(
self
)
->
Cost
:
return
self
.
_cost
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp
```

GroqStreamStructuredResponse
dataclass
Bases:
StreamStructuredResponse
Implementation of
StreamStructuredResponse
for Groq models.
Source code in
pydantic_ai_slim/pydantic_ai/models/groq.py

```
@dataclass
class GroqStreamStructuredResponse(StreamStructuredResponse):
"""Implementation of `StreamStructuredResponse` for Groq models."""
_response: AsyncStream[ChatCompletionChunk]
_delta_tool_calls: dict[int, ChoiceDeltaToolCall]
_timestamp: datetime
_cost: result.Cost
async def __anext__(self) -> None:
chunk = await self._response
```

```
.
__anext__
()
self
.
_cost
=
_map_cost
(
chunk
)
try
:
choice
=
chunk
.
choices
[
0
]
except
IndexError
:
raise
StopAsyncIteration
()
if
choice
.
finish_reason
is
not
None
:
raise
StopAsyncIteration
()
assert
choice
.
delta
.
content
is
None
,
f
'Expected tool calls, got content instead, invalid chunk:
{
chunk
!r}
'
for
new
in
choice
.
delta
.
tool_calls
or
[]:
if
current
:=
self
.
_delta_tool_calls
.
get
(
new
.
index
):
if
```

```
current
.
function
is
None
:
current
.
function
=
new
.
function
elif
new
.
function
is
not
None
:
current
.
function
.
name
=
_utils
.
add_optional
(
current
.
function
.
name
,
new
.
function
.
name
)
current
.
function
.
arguments
=
_utils
.
add_optional
(
current
.
function
.
arguments
,
new
.
function
.
arguments
)
else
:
self
.
_delta_tool_calls
[
new
.
index
]
=
new
```

```
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
ModelStructuredResponse
:
calls
:
list
[
ToolCall
]
=
[]
for
c
in
self
.
_delta_tool_calls
.
values
():
if
f
:=
c
.
function
:
if
f
.
name
is
not
None
and
f
.
arguments
is
not
None
:
calls
.
append
(
ToolCall
.
from_json
(
f
.
name
,
f
.
arguments
,
c
.
id
))
return
ModelStructuredResponse
```

(
calls
,
timestamp
=
self
.
_timestamp
)
def
cost
(
self
)
->
Cost
:
return
self
.
_cost
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp

==
Page: SQL Generation - PydanticAI
URL: https://ai.pydantic.dev/examples/sql-gen/
==

SQL Generation - PydanticAI

pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models
pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Running the Example
Example Code
Introduction
Examples
SQL Generation
Example demonstrating how to use PydanticAI to generate SQL queries based on user input.
Demonstrates:
dynamic system prompt
structured
result_type
result validation
agent dependencies
Running the Example
The resulting SQL is validated by running it as an
EXPLAIN
query on PostgreSQL. To run the example, you first need to run PostgreSQL, e.g. via Docker:
docker
run
--rm
-e
POSTGRES_PASSWORD
=
postgres
-p
54320
:5432
postgres
(we run postgres on port
54320
to avoid conflicts with any other postgres instances you may have running)
With
dependencies installed and environment variables set
, run:
pip
uv
python
-m
pydantic_ai_examples.sql_gen
uv
run
-m
pydantic_ai_examples.sql_gen
or to use a custom prompt:
pip
uv
python
-m
pydantic_ai_examples.sql_gen
"find me errors"
uv
run
-m
pydantic_ai_examples.sql_gen
"find me errors"
This model uses
gemini-1.5-flash
by default since Gemini is good at single shot queries of this kind.
Example Code
sql_gen.py
import
asyncio
import

```
sys
from
collections.abc
import
AsyncGenerator
from
contextlib
import
asynccontextmanager
from
dataclasses
import
dataclass
from
datetime
import
date
from
typing
import
Annotated
,
Any
,
Union
import
asyncpg
import
logfire
from
annotated_types
import
MinLen
from
devtools
import
debug
from
pydantic
import
BaseModel
,
Field
from
typing_extensions
import
TypeAlias
from
pydantic_ai
import
Agent
,
ModelRetry
,
RunContext
# 'if-token-present' means nothing will be sent (and the example will work) if you don't have
logfire configured
logfire
.
configure
(
send_to_logfire
=
'if-token-present'
)
logfire
.
instrument_asyncpg
()
DB_SCHEMA
=
"""
CREATE TABLE records (
created_at timestamptz,
start_timestamp timestamptz,
end_timestamp timestamptz,
trace_id text,
```

```
span_id text,
parent_span_id text,
level log_level,
span_name text,
message text,
attributes_json_schema text,
attributes jsonb,
tags text[],
is_exception boolean,
otel_status_message text,
service_name text
);
"""
@dataclass
class
Deps
:
conn
:
asyncpg
.
Connection
class
Success
(
BaseModel
):
"""Response when SQL could be successfully generated."""
sql_query
:
Annotated
[
str
,
MinLen
(
1
)]
explanation
:
str
=
Field
(
''
,
description
=
'Explanation of the SQL query, as markdown'
)
class
InvalidRequest
(
BaseModel
):
"""Response the user input didn't include enough information to generate SQL."""
error_message
:
str
Response
:
TypeAlias
=
Union
[
Success
,
InvalidRequest
]
agent
=
Agent
(
'gemini-1.5-flash'
,
# Type ignore while we wait for PEP-0747, nonetheless unions will work fine everywhere else
result_type
```

```
=
Response
,
# type: ignore
deps_type
=
Deps
,
)
@agent
.
system_prompt
async
def
system_prompt
()
->
str
:
return
f
"""
\
Given the following PostgreSQL table of records, your job is to
write a SQL query that suits the user's request.
Database schema:
{
DB_SCHEMA
}
today's date =
{
date
.
today
()
}
Example
request: show me records where foobar is false
response: SELECT * FROM records WHERE attributes->>'foobar' = false
Example
request: show me records where attributes include the key "foobar"
response: SELECT * FROM records WHERE attributes ? 'foobar'
Example
request: show me records from yesterday
response: SELECT * FROM records WHERE start_timestamp::date > CURRENT_TIMESTAMP - INTERVAL '1 day'
Example
request: show me error records with the tag "foobar"
response: SELECT * FROM records WHERE level = 'error' and 'foobar' = ANY(tags)
"""
@agent
.
result_validator
async
def
validate_result
(
ctx
:
RunContext
[
Deps
],
result
:
Response
)
->
Response
:
if
isinstance
(
result
,
InvalidRequest
):
return
```

```
result
# gemini often adds extraneous backslashes to SQL
result
.
sql_query
=
result
.
sql_query
.
replace
(
'
\\
'
,
''
)
if
not
result
.
sql_query
.
upper
()
.
startswith
(
'SELECT'
):
raise
ModelRetry
(
'Please create a SELECT query'
)
try
:
await
ctx
.
deps
.
conn
.
execute
(
f
'EXPLAIN
{
result
.
sql_query
}
'
)
except
asyncpg
.
exceptions
.
PostgresError
as
e
:
raise
ModelRetry
(
f
'Invalid query:
{
e
}
'
)
from
e
```

```
else
:
return
result
async
def
main
():
if
len
(
sys
.
argv
)
==
1
:
prompt
=
'show me logs from yesterday, with level "error"'
else
:
prompt
=
sys
.
argv
[
1
]
async
with
database_connect
(
'postgresql://postgres:postgres@localhost:54320'
,
'pydantic_ai_sql_gen'
)
as
conn
:
deps
=
Deps
(
conn
)
result
=
await
agent
.
run
(
prompt
,
deps
=
deps
)
debug
(
result
.
data
)
# pyright: reportUnknownMemberType=false
# pyright: reportUnknownVariableType=false
@asynccontextmanager
async
def
database_connect
(
server_dsn
:
str
```

```
,
database
:
str
)
->
AsyncGenerator
[
Any
,
None
]:
with
logfire
.
span
(
'check and create DB'
):
conn
=
await
asyncpg
.
connect
(
server_dsn
)
try
:
db_exists
=
await
conn
.
fetchval
(
'SELECT 1 FROM pg_database WHERE datname = $1'
,
database
)
if
not
db_exists
:
await
conn
.
execute
(
f
'CREATE DATABASE
{
database
}
'
)
finally
:
await
conn
.
close
()
conn
=
await
asyncpg
.
connect
(
f
'
{
server_dsn
}
/
```

```
{
database
}
'
)
try
:
with
logfire
.
span
(
'create schema'
):
async
with
conn
.
transaction
():
if
not
db_exists
:
await
conn
.
execute
(
"CREATE TYPE log_level AS ENUM ('debug', 'info', 'warning', 'error', 'critical')"
)
await
conn
.
execute
(
DB_SCHEMA
)
yield
conn
finally
:
await
conn
.
close
()
if
__name__
==
'__main__'
:
asyncio
.
run
(
main
())
```


==
Page: Agents - PydanticAI
URL: https://ai.pydantic.dev/agents/
==

Agents - PydanticAI
Skip to content
PydanticAI
Agents
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai
Introduction
Installation & Setup
Getting Help

Table of contents
Introduction
Running Agents
Runs vs. Conversations
Type safe by design
System Prompts
Function Tools
Registering Function Tools via kwarg
Function Tools vs. Structured Results
Function tools and schema
Dynamic Function tools
Reflection and self-correction
Model errors

Introduction
Agents are PydanticAI's primary interface for interacting with LLMs.
In some use cases a single Agent will control an entire application or component,
but multiple agents can also interact to embody more complex workflows.
The
Agent
class has full API documentation, but conceptually you can think of an agent as a container for:
A
system prompt
— a set of instructions for the LLM written by the developer
One or more
retrieval tool

— functions that the LLM may call to get information while generating a response
An optional structured
result type
— the structured datatype the LLM must return at the end of a run
A
dependency
type constraint — system prompt functions, tools and result validators may all use dependencies when they're run
Agents may optionally also have a default
LLM model
associated with them; the model to use can also be specified when running the agent
In typing terms, agents are generic in their dependency and result types, e.g., an agent which required dependencies of type
Foobar
and returned results of type
list
[
str
]
would have type
cAgent[Foobar, list[str]]
. In practice, you shouldn't need to care about this, it should just mean your IDE can tell you when you have the right type, and if you choose to use
static type checking
it should work well with PydanticAI.
Here's a toy example of an agent that simulates a roulette wheel:
roulette_wheel.py

```
from
pydantic_ai
import
Agent
,
RunContext
roulette_agent
=
Agent
(
# (1)!
'openai:gpt-4o'
,
deps_type
=
int
,
result_type
=
bool
,
system_prompt
=
(
'Use the `roulette_wheel` function to see if the '
'customer has won based on the number they provide.'
),
)
@roulette_agent
.
tool
async
def
roulette_wheel
(
ctx
:
RunContext
[
int
],
square
:
int
)
->
str
:
# (2)!
"""check if the square is a winner"""
```

```
return
'winner'
if
square
==
ctx
.
deps
else
'loser'
# Run the agent
success_number
=
18
# (3)!
result
=
roulette_agent
.
run_sync
(
'Put my money on square eighteen'
,
deps
=
success_number
)
print
(
result
.
data
)
# (4)!
#> True
result
=
roulette_agent
.
run_sync
(
'I bet five is the winner'
,
deps
=
success_number
)
print
(
result
.
data
)
#> False
```

Create an agent, which expects an integer dependency and returns a boolean result. This agent will have type
Agent
[
int
,
bool
]
.
Define a tool that checks if the square is a winner. Here
RunContext
is parameterized with the dependency type
int
; if you got the dependency type wrong you'd get a typing error.
In reality, you might want to use a random number here e.g.
random.randint(0, 36)
.
result.data
will be a boolean indicating if the square is a winner. Pydantic performs the result validation, it'll be typed as a
bool
since its type is derived from the
result_type

generic parameter of the agent.
Agents are designed for reuse, like FastAPI Apps
Agents are intended to be instantiated once (frequently as module globals) and reused throughout your application, similar to a small
FastAPI
app or an
APIRouter
.
Running Agents
There are three ways to run an agent:
agent.run()
— a coroutine which returns a
RunResult
containing a completed response
agent.run_sync()
— a plain, synchronous function which returns a
RunResult
containing a completed response (internally, this just calls
loop.run_until_complete(self.run())
)
agent.run_stream()
— a coroutine which returns a
StreamedRunResult
, which contains methods to stream a response as an async iterable
Here's a simple example demonstrating all three:
run_agent.py
```
from
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
)
result_sync
=
agent
.
run_sync
(
'What is the capital of Italy?'
)
print
(
result_sync
.
data
)
#> Rome
async
def
main
():
result
=
await
agent
.
run
(
'What is the capital of France?'
)
print
(
result
.
data
)
#> Paris
async
with
agent
.
run_stream
(
```

'What is the capital of the UK?'
)
as
response
:
print
(
await
response
.
get_data
())
#> London
(This example is complete, it can be run "as is")
You can also pass messages from previous runs to continue a conversation or provide context, as described in
Messages and Chat History
.
jupyter notebooks
If you're running
pydantic-ai
in a jupyter notebook, you might consider using
nest-asyncio
to manage conflicts between event loops that occur between jupyter's event loops and
pydantic-ai
's.
Before you execute any agent runs, do the following:
import
nest_asyncio
nest_asyncio
.
apply
()
Runs vs. Conversations
An agent
run
might represent an entire conversation — there's no limit to how many messages can be exchanged in a single run. However, a
conversation
might also be composed of multiple runs, especially if you need to maintain state between separate interactions or API calls.
Here's an example of a conversation comprised of multiple runs:
conversation_example.py
from
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
)
First run
result1
=
agent
.
run_sync
(
'Who was Albert Einstein?'
)
print
(
result1
.
data
)
#> Albert Einstein was a German-born theoretical physicist.
Second run, passing previous messages
result2
=
agent
.
run_sync
(
'What was his most famous equation?'

,
message_history
=
result1
.
new_messages
(),
(1)!
)
print
(
result2
.
data
)
#> Albert Einstein's most famous equation is (E = mc^2).
Continue the conversation; without
message_history
the model would not know who "his" was referring to.
(This example is complete, it can be run "as is")
Type safe by design
PydanticAI is designed to work well with static type checkers, like mypy and pyright.
Typing is (somewhat) optional
PydanticAI is designed to make type checking as useful as possible for you if you choose to use it, but you don't have to use types everywhere all the time.
That said, because PydanticAI uses Pydantic, and Pydantic uses type hints as the definition for schema and validation, some types (specifically type hints on parameters to tools, and the
result_type
arguments to
Agent
) are used at runtime.
We (the library developers) have messed up if type hints are confusing you more than they're help you, if you find this, please create an
issue
explaining what's annoying you!
In particular, agents are generic in both the type of their dependencies and the type of results they return, so you can use the type hints to ensure you're using the right types.
Consider the following script with type mistakes:
type_mistakes.py
from
dataclasses
import
dataclass
from
pydantic_ai
import
Agent
,
RunContext
@dataclass
class
User
:
name
:
str
agent
=
Agent
(
'test'
,
deps_type
=
User
,
(1)!
result_type
=
bool
,
)
@agent
.
system_prompt
def
add_user_name

(
ctx
:
RunContext
[
str
])
->
str
:
(2)!
return
f
"The user's name is
{
ctx
.
deps
}
."
def
foobar
(
x
:
bytes
)
->
None
:
pass
result
=
agent
.
run_sync
(
'Does their name start with "A"?'
,
deps
=
User
(
'Anne'
))
foobar
(
result
.
data
)
(3)!
The agent is defined as expecting an instance of
User
as
deps
.
But here
add_user_name
is defined as taking a
str
as the dependency, not a
User
.
Since the agent is defined as returning a
bool
, this will raise a type error since
foobar
expects
bytes
.
Running
mypy
on this will give the following output:
➤
uv
run

mypy
type_mistakes.py
type_mistakes.py:18:
error:
Argument
1
to
"system_prompt"
of
"Agent"
has
incompatible
type
"Callable[[RunContext[str]], str]"
;
expected
"Callable[[RunContext[User]], str]"
[
arg-type
]
type_mistakes.py:28:
error:
Argument
1
to
"foobar"
has
incompatible
type
"bool"
;
expected
"bytes"
[
arg-type
]
Found
2
errors
in
1
file
(
checked
1
source
file
)
Running
pyright
would identify the same issues.
System Prompts
System prompts might seem simple at first glance since they're just strings (or sequences of strings that are concatenated), but crafting the right system prompt is key to getting the model to behave as you want.
Generally, system prompts fall into two categories:
Static system prompts
: These are known when writing the code and can be defined via the
system_prompt
parameter of the
Agent
constructor
.
Dynamic system prompts
: These depend in some way on context that isn't known until runtime, and should be defined via functions decorated with
@agent.system_prompt
.
You can add both to a single agent; they're appended in the order they're defined at runtime.
Here's an example using both types of system prompts:
system_prompts.py
from
datetime
import
date
from
pydantic_ai

```
import
Agent
,
RunContext
agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
str
,
# (1)!
system_prompt
=
"Use the customer's name while replying to them."
,
# (2)!
)
@agent
.
system_prompt
# (3)!
def
add_the_users_name
(
ctx
:
RunContext
[
str
])
->
str
:
return
f
"The user's named is
{
ctx
.
deps
}
."
@agent
.
system_prompt
def
add_the_date
()
->
str
:
# (4)!
return
f
'The date is
{
date
.
today
()
}
.'
result
=
agent
.
run_sync
(
'What is the date?'
,
deps
=
'Frank'
```

)
print
(
result
.
data
)
#> Hello Frank, the date today is 2032-01-02.
The agent expects a string dependency.
Static system prompt defined at agent creation time.
Dynamic system prompt defined via a decorator with
RunContext
, this is called just after
run_sync
, not when the agent is created, so can benefit from runtime information like the dependencies used on that run.
Another dynamic system prompt, system prompts don't have to have the
RunContext
parameter.
(This example is complete, it can be run "as is")
Function Tools
Function tools provide a mechanism for models to retrieve extra information to help them generate a response.
They're useful when it is impractical or impossible to put all the context an agent might need into the system prompt, or when you want to make agents' behavior more deterministic or reliable by deferring some of the logic required to generate a response to another (not necessarily AI-powered) tool.
Function tools vs. RAG
Function tools are basically the "R" of RAG (Retrieval-Augmented Generation) — they augment what the model can do by letting it request extra information.
The main semantic difference between PydanticAI Tools and RAG is RAG is synonymous with vector search, while PydanticAI tools are more general-purpose. (Note: we may add support for vector search functionality in the future, particularly an API for generating embeddings. See
#58
)
There are a number of ways to register tools with an agent:
via the
@agent.tool
decorator — for tools that need access to the agent
context
via the
@agent.tool_plain
decorator — for tools that do not need access to the agent
context
via the
tools
keyword argument to
Agent
which can take either plain functions, or instances of
Tool
@agent.tool
is considered the default decorator since in the majority of cases tools will need access to the agent context.
Here's an example using both:
dice_game.py
import
random
from
pydantic_ai
import
Agent
,
RunContext
agent
=
Agent
(
'gemini-1.5-flash'
,
(1)!
deps_type
=
str
,
(2)!
system_prompt
=

```
(
"You're a dice game, you should roll the die and see if the number "
"you get back matches the user's guess. If so, tell them they're a winner. "
"Use the player's name in the response."
),
)
@agent
.
tool_plain
# (3)!
def
roll_die
()
->
str
:
"""Roll a six-sided die and return the result."""
return
str
(
random
.
randint
(
1
,
6
))
@agent
.
tool
# (4)!
def
get_player_name
(
ctx
:
RunContext
[
str
])
->
str
:
"""Get the player's name."""
return
ctx
.
deps
dice_result
=
agent
.
run_sync
(
'My guess is 4'
,
deps
=
'Anne'
)
# (5)!
print
(
dice_result
.
data
)
#> Congratulations Anne, you guessed correctly! You're a winner!
```

This is a pretty simple task, so we can use the fast and cheap Gemini flash model.
We pass the user's name as the dependency, to keep things simple we use just the name as a string as the dependency.
This tool doesn't need any context, it just returns a random number. You could probably use a dynamic system prompt in this case.
This tool needs the player's name, so it uses
RunContext
to access dependencies which are just the player's name in this case.

Run the agent, passing the player's name as the dependency.
(This example is complete, it can be run "as is")
Let's print the messages from that game to see what happened:
dice_game_messages.py

```
from
dice_game
import
dice_result
print
(
dice_result
.
all_messages
())
"""
[
SystemPrompt(
content="You're a dice game, you should roll the die and see if the number you get back matches the user's guess. If so, tell them they're a winner. Use the player's name in the response.",
role='system',
),
UserPrompt(
content='My guess is 4',
timestamp=datetime.datetime(...),
role='user',
),
ModelStructuredResponse(
calls=[
ToolCall(
tool_name='roll_die', args=ArgsDict(args_dict={}), tool_call_id=None
)
],
timestamp=datetime.datetime(...),
role='model-structured-response',
),
ToolReturn(
tool_name='roll_die',
content='4',
tool_call_id=None,
timestamp=datetime.datetime(...),
role='tool-return',
),
ModelStructuredResponse(
calls=[
ToolCall(
tool_name='get_player_name',
args=ArgsDict(args_dict={}),
tool_call_id=None,
)
],
timestamp=datetime.datetime(...),
role='model-structured-response',
),
ToolReturn(
tool_name='get_player_name',
content='Anne',
tool_call_id=None,
timestamp=datetime.datetime(...),
role='tool-return',
),
ModelTextResponse(
content="Congratulations Anne, you guessed correctly! You're a winner!",
timestamp=datetime.datetime(...),
role='model-text-response',
),
]
"""
```

We can represent this with a diagram:

```
sequenceDiagram
    participant Agent
    participant LLM

    Note over Agent: Send prompts
    Agent ->> LLM: System: "You're a dice game..."<br>User: "My guess is 4"
    activate LLM
    Note over LLM: LLM decides to use<br>a tool
```

```
    LLM ->> Agent: Call tool<br>roll_die()
    deactivate LLM
    activate Agent
    Note over Agent: Rolls a six-sided die

    Agent -->> LLM: ToolReturn<br>"4"
    deactivate Agent
    activate LLM
    Note over LLM: LLM decides to use<br>another tool

    LLM ->> Agent: Call tool<br>get_player_name()
    deactivate LLM
    activate Agent
    Note over Agent: Retrieves player name
    Agent -->> LLM: ToolReturn<br>"Anne"
    deactivate Agent
    activate LLM
    Note over LLM: LLM constructs final response

    LLM ->> Agent: ModelTextResponse<br>"Congratulations Anne, ..."
    deactivate LLM
    Note over Agent: Game session complete
```

Registering Function Tools via kwarg

As well as using the decorators, we can register tools via the
tools
argument to the
Agent
constructor
. This is useful when you want to re-use tools, and can also give more fine-grained control over the tools.

dice_game_tool_kwarg.py

```
import
random
from
pydantic_ai
import
Agent
,
RunContext
,
Tool
def
roll_die
()
->
str
:
"""Roll a six-sided die and return the result."""
return
str
(
random
.
randint
(
1
,
6
))
def
get_player_name
(
ctx
:
RunContext
[
str
])
->
str
:
"""Get the player's name."""
return
ctx
.
deps
agent_a
```

```
=
Agent
(
'gemini-1.5-flash'
,
deps_type
=
str
,
tools
=
[
roll_die
,
get_player_name
],
# (1)!
)
agent_b
=
Agent
(
'gemini-1.5-flash'
,
deps_type
=
str
,
tools
=
[
# (2)!
Tool
(
roll_die
,
takes_ctx
=
False
),
Tool
(
get_player_name
,
takes_ctx
=
True
),
],
)
dice_result
=
agent_b
.
run_sync
(
'My guess is 4'
,
deps
=
'Anne'
)
print
(
dice_result
.
data
)
#> Congratulations Anne, you guessed correctly! You're a winner!
```
The simplest way to register tools via the
Agent
constructor is to pass a list of functions, the function signature is inspected to determine if the tool takes
RunContext
.
agent_a
and

agent_b
are identical — but we can use
Tool
to reuse tool definitions and give more fine-grained control over how tools are defined, e.g. setting their name or description, or using a custom
prepare
method.
(This example is complete, it can be run "as is")
Function Tools vs. Structured Results
As the name suggests, function tools use the model's "tools" or "functions" API to let the model know what is available to call. Tools or functions are also used to define the schema(s) for structured responses, thus a model might have access to many tools, some of which call function tools while others end the run and return a result.
Function tools and schema
Function parameters are extracted from the function signature, and all parameters except
RunContext
are used to build the schema for that tool call.
Even better, PydanticAI extracts the docstring from functions and (thanks to
griffe
) extracts parameter descriptions from the docstring and adds them to the schema.
Griffe supports
extracting parameter descriptions from
google
,
numpy
and
sphinx
style docstrings, and PydanticAI will infer the format to use based on the docstring. We plan to add support in the future to explicitly set the style to use, and warn/error if not all parameters are documented; see
#59
.
To demonstrate a tool's schema, here we use
FunctionModel
to print the schema a model would receive:
tool_schema.py
```
from
pydantic_ai
import
Agent
from
pydantic_ai.messages
import
Message
,
ModelAnyResponse
,
ModelTextResponse
from
pydantic_ai.models.function
import
AgentInfo
,
FunctionModel
agent
=
Agent
()
@agent
.
tool_plain
def
foobar
(
a
:
int
,
b
:
str
,
c
:
dict
[
str
```

```
,
list
[
float
]])
->
str
:
"""Get me foobar.
Args:
a: apple pie
b: banana cake
c: carrot smoothie
"""
return
f
'
{
a
}
{
b
}
{
c
}
'
def
print_schema
(
messages
:
list
[
Message
],
info
:
AgentInfo
)
->
ModelAnyResponse
:
tool
=
info
.
function_tools
[
0
]
print
(
tool
.
description
)
#> Get me foobar.
print
(
tool
.
parameters_json_schema
)
"""
{
'description': 'Get me foobar.',
'properties': {
'a': {'description': 'apple pie', 'title': 'A', 'type': 'integer'},
'b': {'description': 'banana cake', 'title': 'B', 'type': 'string'},
'c': {
'additionalProperties': {'items': {'type': 'number'}, 'type': 'array'},
'description': 'carrot smoothie',
'title': 'C',
'type': 'object',
},
},
```

```
'required': ['a', 'b', 'c'],
'type': 'object',
'additionalProperties': False,
}
"""
return
ModelTextResponse
(
content
=
'foobar'
)
agent
.
run_sync
(
'hello'
,
model
=
FunctionModel
(
print_schema
))
```

(This example is complete, it can be run "as is")

The return type of tool can be anything which Pydantic can serialize to JSON as some models (e.g. Gemini) support semi-structured return values, some expect text (OpenAI) but seem to be just as good at extracting meaning from the data. If a Python object is returned and the model expects a string, the value will be serialized to JSON.

If a tool has a single parameter that can be represented as an object in JSON schema (e.g. dataclass, TypedDict, pydantic model), the schema for the tool is simplified to be just that object.

Here's an example, we use
TestModel.agent_model_function_tools
to inspect the tool schema that would be passed to the model.

single_parameter_tool.py

```
from
pydantic
import
BaseModel
from
pydantic_ai
import
Agent
from
pydantic_ai.models.test
import
TestModel
agent
=
Agent
()
class
Foobar
(
BaseModel
):
"""This is a Foobar"""
x
:
int
y
:
str
z
:
float
=
3.14
@agent
.
tool_plain
def
foobar
(
f
:
Foobar
```

)
->
str
:
return
str
(
f
)
test_model
=
TestModel
()
result
=
agent
.
run_sync
(
'hello'
,
model
=
test_model
)
print
(
result
.
data
)
#> {"foobar":"x=0 y='a' z=3.14"}
print
(
test_model
.
agent_model_function_tools
)
"""
[
ToolDefinition(
name='foobar',
description='',
parameters_json_schema={
'description': 'This is a Foobar',
'properties': {
'x': {'title': 'X', 'type': 'integer'},
'y': {'title': 'Y', 'type': 'string'},
'z': {'default': 3.14, 'title': 'Z', 'type': 'number'},
},
'required': ['x', 'y'],
'title': 'Foobar',
'type': 'object',
},
outer_typed_dict_key=None,
)
]
"""
(This example is complete, it can be run "as is")
Dynamic Function tools
Tools can optionally be defined with another function:
prepare
, which is called at each step of a run to
customize the definition of the tool passed to the model, or omit the tool completely from that
step.
A
prepare
method can be registered via the
prepare
kwarg to any of the tool registration mechanisms:
@agent.tool
decorator
@agent.tool_plain
decorator
Tool
dataclass
The

prepare
method, should be of type
ToolPrepareFunc
, a function which takes
RunContext
and a pre-built
ToolDefinition
, and should either return that
ToolDefinition
with or without modifying it, return a new
ToolDefinition
, or return
None
to indicate this tools should not be registered for that step.
Here's a simple
prepare
method that only includes the tool if the value of the dependency is
42
.
As with the previous example, we use
TestModel
to demonstrate the behavior without calling a real model.
tool_only_if_42.py

```
from
typing
import
Union
from
pydantic_ai
import
Agent
,
RunContext
from
pydantic_ai.tools
import
ToolDefinition
agent
=
Agent
(
'test'
)
async
def
only_if_42
(
ctx
:
RunContext
[
int
],
tool_def
:
ToolDefinition
)
->
Union
[
ToolDefinition
,
None
]:
if
ctx
.
deps
==
42
:
return
tool_def
@agent
.
tool
(
```

prepare
=
only_if_42
)
def
hitchhiker
(
ctx
:
RunContext
[
int
],
answer
:
str
)
->
str
:
return
f
'
{
ctx
.
deps
}
{
answer
}
'
result
=
agent
.
run_sync
(
'testing...'
,
deps
=
41
)
print
(
result
.
data
)
#> success (no tool calls)
result
=
agent
.
run_sync
(
'testing...'
,
deps
=
42
)
print
(
result
.
data
)
#> {"hitchhiker":"42 a"}
(This example is complete, it can be run "as is")
Here's a more complex example where we change the description of the
name
parameter to based on the value of
deps
For the sake of variation, we create this tool using the
Tool

dataclass.
customize_name.py

```
from
__future__
import
annotations
from
typing
import
Literal
from
pydantic_ai
import
Agent
,
RunContext
from
pydantic_ai.models.test
import
TestModel
from
pydantic_ai.tools
import
Tool
,
ToolDefinition
def
greet
(
name
:
str
)
->
str
:
return
f
'hello
{
name
}
'
async
def
prepare_greet
(
ctx
:
RunContext
[
Literal
[
'human'
,
'machine'
]],
tool_def
:
ToolDefinition
)
->
ToolDefinition
|
None
:
d
=
f
'Name of the
{
ctx
.
deps
}
to greet.'
tool_def
```

```
.
parameters_json_schema
[
'properties'
][
'name'
][
'description'
]
=
d
return
tool_def
greet_tool
=
Tool
(
greet
,
prepare
=
prepare_greet
)
test_model
=
TestModel
()
agent
=
Agent
(
test_model
,
tools
=
[
greet_tool
],
deps_type
=
Literal
[
'human'
,
'machine'
])
result
=
agent
.
run_sync
(
'testing...'
,
deps
=
'human'
)
print
(
result
.
data
)
#> {"greet":"hello a"}
print
(
test_model
.
agent_model_function_tools
)
"""
[
ToolDefinition(
name='greet',
description='',
parameters_json_schema={
```

```
'properties': {
'name': {
'title': 'Name',
'type': 'string',
'description': 'Name of the human to greet.',
}
},
'required': ['name'],
'type': 'object',
'additionalProperties': False,
},
outer_typed_dict_key=None,
)
]
"""
```

(This example is complete, it can be run "as is")

Reflection and self-correction

Validation errors from both function tool parameter validation and
structured result validation
can be passed back to the model with a request to retry.
You can also raise
ModelRetry
from within a
tool
or
result validator function
to tell the model it should retry generating a response.
The default retry count is
1
but can be altered for the
entire agent
, a
specific tool
, or a
result validator
.
You can access the current retry count from within a tool or result validator via
ctx.retry
.
Here's an example:
tool_retry.py

```
from
fake_database
import
DatabaseConn
from
pydantic
import
BaseModel
from
pydantic_ai
import
Agent
,
RunContext
,
ModelRetry
class
ChatResult
(
BaseModel
):
user_id
:
int
message
:
str
agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
DatabaseConn
```

```
,
result_type
=
ChatResult
,
)
@agent
.
tool
(
retries
=
2
)
def
get_user_by_name
(
ctx
:
RunContext
[
DatabaseConn
],
name
:
str
)
->
int
:
"""Get a user's ID from their full name."""
print
(
name
)
#> John
#> John Doe
user_id
=
ctx
.
deps
.
users
.
get
(
name
=
name
)
if
user_id
is
None
:
raise
ModelRetry
(
f
'No user found with name
{
name
!r}
, remember to provide their full name'
)
return
user_id
result
=
agent
.
run_sync
(
'Send a message to John Doe asking for coffee next week'
,
deps
```

```
=
DatabaseConn
()
)
print
(
result
.
data
)
"""
user_id=123 message='Hello John, would you be free for coffee sometime next week? Let me know what works for you!'
"""
```

Model errors

If models behave unexpectedly (e.g., the retry limit is exceeded, or their API returns 503), agent runs will raise UnexpectedModelBehavior.

In these cases, agent.last_run_messages can be used to access the messages exchanged during the run to help diagnose the issue.

```
from
pydantic_ai
import
Agent
,
ModelRetry
,
UnexpectedModelBehavior
agent
=
Agent
(
'openai:gpt-4o'
)
@agent
.
tool_plain
def
calc_volume
(
size
:
int
)
->
int
:
# (1)!
if
size
==
42
:
return
size
**
3
else
:
raise
ModelRetry
(
'Please try again.'
)
try
:
result
=
agent
.
run_sync
(
'Please get me the volume of a box with size 6.'
)
```

```
except
UnexpectedModelBehavior
as
e
:
print
(
'An error occurred:'
,
e
)
#> An error occurred: Tool exceeded max retries count of 1
print
(
'cause:'
,
repr
(
e
.
__cause__
))
#> cause: ModelRetry('Please try again.')
print
(
'messages:'
,
agent
.
last_run_messages
)
"""
messages:
[
UserPrompt(
content='Please get me the volume of a box with size 6.',
timestamp=datetime.datetime(...),
role='user',
),
ModelStructuredResponse(
calls=[
ToolCall(
tool_name='calc_volume',
args=ArgsDict(args_dict={'size': 6}),
tool_call_id=None,
)
],
timestamp=datetime.datetime(...),
role='model-structured-response',
),
RetryPrompt(
content='Please try again.',
tool_name='calc_volume',
tool_call_id=None,
timestamp=datetime.datetime(...),
role='retry-prompt',
),
ModelStructuredResponse(
calls=[
ToolCall(
tool_name='calc_volume',
args=ArgsDict(args_dict={'size': 6}),
tool_call_id=None,
)
],
timestamp=datetime.datetime(...),
role='model-structured-response',
),
]
"""
else
:
print
(
result
.
data
```

```
)
```
1. Define a tool that will raise
ModelRetry
repeatedly in this case.
(This example is complete, it can be run "as is")

==
Page: Installation & Setup - PydanticAI
URL: https://ai.pydantic.dev/install/
==

Installation & Setup - PydanticAI
Skip to content
PydanticAI
Installation & Setup
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Use with Pydantic Logfire
Running Examples
Slim Install
Model Configuration
OpenAI
Environment variable
api_key argument
Custom OpenAI Client
Gemini
Environment variable
api_key argument
Gemini via VertexAI
application default credentials
service account
Customising region
Anthropic
Environment variable
api_key argument
Groq
Environment variable
api_key argument
Ollama
Installation & Setup
PydanticAI is available on PyPI as
pydantic-ai
so installation is as simple as:
pip
uv
pip
install
pydantic-ai
uv
add
pydantic-ai
(Requires Python 3.9+)
This installs the
pydantic_ai
package, core dependencies, and libraries required to use the following LLM APIs:
OpenAI API
Google VertexAI API
for Gemini models
Anthropic API
Groq API
Use with Pydantic Logfire
PydanticAI has an excellent (but completely optional) integration with
Pydantic Logfire
to help you view and understand agent runs.
To use Logfire with PydanticAI, install
pydantic-ai
or
pydantic-ai-slim
with the
logfire
optional group:
pip
uv
pip
install
'pydantic-ai[logfire]'
uv
add
'pydantic-ai[logfire]'
From there, follow the
Logfire setup docs
to configure Logfire.
Running Examples
We distributes the
pydantic_ai_examples
directory as a separate PyPI package (
pydantic-ai-examples

) to make examples extremely easy to customize and run.
To install examples, use the
examples
optional group:
pip
uv
pip
install
'pydantic-ai[examples]'
uv
add
'pydantic-ai[examples]'
To run the examples, follow instructions in the
examples docs
.
Slim Install
If you know which model you're going to use and want to avoid installing superfluous packages, you can use the
pydantic-ai-slim
package.
If you're using just
OpenAIModel
, run:
pip
uv
pip
install
'pydantic-ai-slim[openai]'
uv
add
'pydantic-ai-slim[openai]'
If you're using just
GeminiModel
(Gemini via the
generativelanguage.googleapis.com
API) no extra dependencies are required, run:
pip
uv
pip
install
pydantic-ai-slim
uv
add
pydantic-ai-slim
If you're using just
VertexAIModel
, run:
pip
uv
pip
install
'pydantic-ai-slim[vertexai]'
uv
add
'pydantic-ai-slim[vertexai]'
If you're just using
Anthropic
, run:
pip
uv
pip
install
'pydantic-ai-slim[anthropic]'
uv
add
'pydantic-ai-slim[anthropic]'
To use just
GroqModel
, run:
pip
uv
pip
install
'pydantic-ai-slim[groq]'
uv
add
'pydantic-ai-slim[groq]'

You can install dependencies for multiple models and use cases, for example:

```
pip
uv
pip
install
'pydantic-ai-slim[openai,vertexai,logfire]'
uv
add
'pydantic-ai-slim[openai,vertexai,logfire]'
```

Model Configuration

To use hosted commercial models, you need to configure your local environment with the appropriate API keys.

OpenAI

To use OpenAI through their main API, go to
platform.openai.com
and follow your nose until you find the place to generate an API key.

Environment variable

Once you have the API key, you can set it as an environment variable:

```
export
OPENAI_API_KEY
=
'your-api-key'
```

You can then use
OpenAIModel
by name:

openai_model_by_name.py

```
from
pydantic_ai
import
Agent
agent
=
Agent
(
'openai:gpt-4o'
)
...
```

Or initialise the model directly with just the model name:

openai_model_init.py

```
from
pydantic_ai
import
Agent
from
pydantic_ai.models.openai
import
OpenAIModel
model
=
OpenAIModel
(
'gpt-4o'
)
agent
=
Agent
(
model
)
...
```

api_key
argument

If you don't want to or can't set the environment variable, you can pass it at runtime via the
api_key
argument
:

openai_model_api_key.py

```
from
pydantic_ai
import
Agent
from
pydantic_ai.models.openai
import
OpenAIModel
model
=
```

OpenAIModel
(
'gpt-4o'
,
api_key
=
'your-api-key'
)
agent
=
Agent
(
model
)
...
Custom OpenAI Client
OpenAIModel
also accepts a custom
AsyncOpenAI
client via the
openai_client
parameter
,
so you can customise the
organization
,
project
,
base_url
etc. as defined in the
OpenAI API docs
.
You could also use the
AsyncAzureOpenAI
client to use the Azure OpenAI API.
openai_azure.py
from
openai
import
AsyncAzureOpenAI
from
pydantic_ai
import
Agent
from
pydantic_ai.models.openai
import
OpenAIModel
client
=
AsyncAzureOpenAI
(
azure_endpoint
=
'...'
,
api_version
=
'2024-07-01-preview'
,
api_key
=
'your-api-key'
,
)
model
=
OpenAIModel
(
'gpt-4o'
,
openai_client
=
client
)
agent
=

Agent
(
model
)
...
Gemini
GeminiModel
let's you use the Google's Gemini models through their
generativelanguage.googleapis.com
API.
GeminiModelName
contains a list of available Gemini models that can be used through this interface.
For prototyping only
Google themselves refer to this API as the "hobby" API, I've received 503 responses from it a number of times.
The API is easy to use and useful for prototyping and simple demos, but I would not rely on it in production.
If you want to run Gemini models in production, you should use the
VertexAI API
described below.
To use
GeminiModel
, go to
aistudio.google.com
and follow your nose until you find the place to generate an API key.
Environment variable
Once you have the API key, you can set it as an environment variable:
export
GEMINI_API_KEY
=
your-api-key
You can then use
GeminiModel
by name:
gemini_model_by_name.py
from
pydantic_ai
import
Agent
agent
=
Agent
(
'gemini-1.5-flash'
)
...
Or initialise the model directly with just the model name:
gemini_model_init.py
from
pydantic_ai
import
Agent
from
pydantic_ai.models.gemini
import
GeminiModel
model
=
GeminiModel
(
'gemini-1.5-flash'
)
agent
=
Agent
(
model
)
...
api_key
argument
If you don't want to or can't set the environment variable, you can pass it at runtime via the
api_key
argument
:
gemini_model_api_key.py
from

pydantic_ai
import
Agent
from
pydantic_ai.models.gemini
import
GeminiModel
model
=
GeminiModel
(
'gemini-1.5-flash'
,
api_key
=
'your-api-key'
)
agent
=
Agent
(
model
)
...
Gemini via VertexAI
To run Google's Gemini models in production, you should use
VertexAIModel
which uses the
*-aiplatform.googleapis.com
API.
GeminiModelName
contains a list of available Gemini models that can be used through this interface.
This interface has a number of advantages over
generativelanguage.googleapis.com
documented above:
The VertexAI API is more reliably and marginally lower latency in our experience.
You can
purchase provisioned throughput
with VertexAI to guarantee capacity.
If you're running PydanticAI inside GCP, you don't need to set up authentication, it should "just work".
You can decide which region to use, which might be important from a regulatory perspective, and might improve latency.
The big disadvantage is that for local development you may need to create and configure a "service account", which I've found extremely painful to get right in the past.
Whichever way you authenticate, you'll need to have VertexAI enabled in your GCP account.
application default credentials
Luckily if you're running PydanticAI inside GCP, or you have the
gcloud
CLI
installed and configured, you should be able to use
VertexAIModel
without any additional setup.
To use
VertexAIModel
, with
application default credentials
configured (e.g. with
gcloud
), you can simply use:
vertexai_application_default_credentials.py
from
pydantic_ai
import
Agent
from
pydantic_ai.models.vertexai
import
VertexAIModel
model
=
VertexAIModel
(
'gemini-1.5-flash'
)
agent
=

Agent
(
model
)
...
Internally this uses
google.auth.default()
from the
google-auth
package to obtain credentials.
Won't fail until
agent.run()
Because
google.auth.default()
requires network requests and can be slow, it's not run until you call
agent.run()
. Meaning any configuration or permissions error will only be raised when you try to use the model.
To for this check to be run, call
await model.agent_model({}, False, None)
.
You may also need to pass the
project_id
argument to
VertexAIModel
if application default credentials don't set a project, if you pass
project_id
and it conflicts with the project set by application default credentials, an error is raised.
service account
If instead of application default credentials, you want to authenticate with a service account, you'll need to create a service account, add it to your GCP project (note: AFAIK this step is necessary even if you created the service account within the project), give that service account the "Vertex AI Service Agent" role, and download the service account JSON file.
Once you have the JSON file, you can use it thus:
vertexai_service_account.py

```
from
pydantic_ai
import
Agent
from
pydantic_ai.models.vertexai
import
VertexAIModel
model
=
VertexAIModel
(
'gemini-1.5-flash'
,
service_account_file
=
'path/to/service-account.json'
,
)
agent
=
Agent
(
model
)
...
```

Customising region
Whichever way you authenticate, you can specify which region requests will be sent to via the
region
argument
.
Using a region close to your application can improve latency and might be important from a regulatory perspective.
vertexai_region.py

```
from
pydantic_ai
import
Agent
from
pydantic_ai.models.vertexai
import
VertexAIModel
model
```

=
VertexAIModel
(
'gemini-1.5-flash'
,
region
=
'asia-east1'
)
agent
=
Agent
(
model
)
...
VertexAiRegion
contains a list of available regions.
Anthropic
To use
Anthropic
through their API, go to
console.anthropic.com/settings/keys
to generate an API key.
AnthropicModelName
contains a list of available Anthropic models.
Environment variable
Once you have the API key, you can set it as an environment variable:
export
ANTHROPIC_API_KEY
=
'your-api-key'
You can then use
AnthropicModel
by name:
anthropic_model_by_name.py
from
pydantic_ai
import
Agent
agent
=
Agent
(
'claude-3-5-sonnet-latest'
)
...
Or initialise the model directly with just the model name:
anthropic_model_init.py
from
pydantic_ai
import
Agent
from
pydantic_ai.models.anthropic
import
AnthropicModel
model
=
AnthropicModel
(
'claude-3-5-sonnet-latest'
)
agent
=
Agent
(
model
)
...
api_key
argument
If you don't want to or can't set the environment variable, you can pass it at runtime via the
api_key
argument
:
anthropic_model_api_key.py

from
pydantic_ai
import
Agent
from
pydantic_ai.models.anthropic
import
AnthropicModel
model
=
AnthropicModel
(
'claude-3-5-sonnet-latest'
,
api_key
=
'your-api-key'
)
agent
=
Agent
(
model
)
...
Groq
To use
Groq
through their API, go to
console.groq.com/keys
and follow your nose until you find the place to generate an API key.
GroqModelName
contains a list of available Groq models.
Environment variable
Once you have the API key, you can set it as an environment variable:
export
GROQ_API_KEY
=
'your-api-key'
You can then use
GroqModel
by name:
groq_model_by_name.py
from
pydantic_ai
import
Agent
agent
=
Agent
(
'groq:llama-3.1-70b-versatile'
)
...
Or initialise the model directly with just the model name:
groq_model_init.py
from
pydantic_ai
import
Agent
from
pydantic_ai.models.groq
import
GroqModel
model
=
GroqModel
(
'llama-3.1-70b-versatile'
)
agent
=
Agent
(
model
)
...

api_key
argument
If you don't want to or can't set the environment variable, you can pass it at runtime via the
api_key
argument
:
groq_model_api_key.py
from
pydantic_ai
import
Agent
from
pydantic_ai.models.groq
import
GroqModel
model
=
GroqModel
(
'llama-3.1-70b-versatile'
,
api_key
=
'your-api-key'
)
agent
=
Agent
(
model
)
...
Ollama
To use
Ollama
, you must first download the Ollama client, and then download a model.
You must also ensure the Ollama server is running when trying to make requests to it. For more information, please see the
Ollama documentation

==
Page: Examples - PydanticAI
URL: https://ai.pydantic.dev/examples/
==

Examples - PydanticAI
Skip to content
PydanticAI
Examples
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

API Reference
API Reference

Table of contents

Introduction
Examples
Examples
Examples of how to use PydanticAI and what it can do.
Usage
These examples are distributed with
pydantic-ai
so you can run them either by cloning the
pydantic-ai repo
or by simply installing
pydantic-ai
from PyPI with
pip
or
uv
.
Installing required dependencies
Either way you'll need to install extra dependencies to run some examples, you just need to install
the
examples
optional dependency group.
If you've installed
pydantic-ai
via pip/uv, you can install the extra dependencies with:
pip
uv
pip
install
'pydantic-ai[examples]'
uv
add
'pydantic-ai[examples]'
If you clone the repo, you should instead use
uv sync --extra examples
to install extra dependencies.
Setting model environment variables
These examples will need you to set up authentication with one or more of the LLMs, see the
model configuration
docs for details on how to do this.
TL;DR: in most cases you'll need to set one of the following environment variables:
OpenAI
Google Gemini
export
OPENAI_API_KEY
=
your-api-key
export
GEMINI_API_KEY
=
your-api-key
Running Examples
To run the examples (this will work whether you installed
pydantic_ai
, or cloned the repo), run:
pip
uv

python
-m
pydantic_ai_examples.<example_module_name>
uv
run
-m
pydantic_ai_examples.<example_module_name>
For examples, to run the very simple
pydantic_model
example:
pip
uv
python
-m
pydantic_ai_examples.pydantic_model
uv
run
-m
pydantic_ai_examples.pydantic_model
If you like one-liners and you're using uv, you can run a pydantic-ai example with zero setup:
OPENAI_API_KEY
=
'your-api-key'
\
uv
run
--with
'pydantic-ai[examples]'
\
-m
pydantic_ai_examples.pydantic_model
You'll probably want to edit examples in addition to just running them. You can copy the examples to a new directory with:
pip
uv
python
-m
pydantic_ai_examples
--copy-to
examples/
uv
run
-m
pydantic_ai_examples
--copy-to
examples/

==
Page: Weather agent - PydanticAI
URL: https://ai.pydantic.dev/examples/weather-agent/
==

Weather agent - PydanticAI

Weather agent
Weather agent

Weather agent
Example of PydanticAI with multiple tools which the LLM needs to call in turn to answer a question.
Demonstrates:
tools
agent dependencies
streaming text responses
In this case the idea is a "weather" agent — the user can ask for the weather in multiple locations, the agent will use the
get_lat_lng
tool to get the latitude and longitude of the locations, then use
the
get_weather
tool to get the weather for those locations.
Running the Example
To run this example properly, you might want to add two extra API keys
(Note if either key is missing, the code will fall back to dummy data, so they're not required)
:
A weather API key from
tomorrow.io
set via
WEATHER_API_KEY
A geocoding API key from
geocode.maps.co
set via
GEO_API_KEY
With
dependencies installed and environment variables set
, run:
pip
uv
```
python
-m
pydantic_ai_examples.weather_agent
uv
run
-m
pydantic_ai_examples.weather_agent
```
Example Code
pydantic_ai_examples/weather_agent.py
```
from
__future__
import
annotations
as
_annotations
```

```
import
asyncio
import
os
from
dataclasses
import
dataclass
from
typing
import
Any
import
logfire
from
devtools
import
debug
from
httpx
import
AsyncClient
from
pydantic_ai
import
Agent
,
ModelRetry
,
RunContext
# 'if-token-present' means nothing will be sent (and the example will work) if you don't have
logfire configured
logfire
.
configure
(
send_to_logfire
=
'if-token-present'
)
@dataclass
class
Deps
:
client
:
AsyncClient
weather_api_key
:
str
|
None
geo_api_key
:
str
|
None
weather_agent
=
Agent
(
'openai:gpt-4o'
,
system_prompt
=
'Be concise, reply with one sentence.'
,
deps_type
=
Deps
,
retries
=
2
,
)
@weather_agent
```

```
.
tool
async
def
get_lat_lng
(
ctx
:
RunContext
[
Deps
],
location_description
:
str
)
->
dict
[
str
,
float
]:
"""Get the latitude and longitude of a location.
Args:
ctx: The context.
location_description: A description of a location.
"""
if
ctx
.
deps
.
geo_api_key
is
None
:
# if no API key is provided, return a dummy response (London)
return
{
'lat'
:
51.1
,
'lng'
:
-
0.1
}
params
=
{
'q'
:
location_description
,
'api_key'
:
ctx
.
deps
.
geo_api_key
,
}
with
logfire
.
span
(
'calling geocode API'
,
params
=
params
)
as
```

```
span
:
r
=
await
ctx
.
deps
.
client
.
get
(
'https://geocode.maps.co/search'
,
params
=
params
)
r
.
raise_for_status
()
data
=
r
.
json
()
span
.
set_attribute
(
'response'
,
data
)
if
data
:
return
{
'lat'
:
data
[
0
][
'lat'
],
'lng'
:
data
[
0
][
'lon'
]}
else
:
raise
ModelRetry
(
'Could not find the location'
)
@weather_agent
.
tool
async
def
get_weather
(
ctx
:
RunContext
[
Deps
```

```
],
lat
:
float
,
lng
:
float
)
->
dict
[
str
,
Any
]:
"""Get the weather at a location.
Args:
ctx: The context.
lat: Latitude of the location.
lng: Longitude of the location.
"""
if
ctx
.
deps
.
weather_api_key
is
None
:
# if no API key is provided, return a dummy response
return
{
'temperature'
:
'21 °C'
,
'description'
:
'Sunny'
}
params
=
{
'apikey'
:
ctx
.
deps
.
weather_api_key
,
'location'
:
f
'
{
lat
}
,
{
lng
}
'
,
'units'
:
'metric'
,
}
with
logfire
.
span
(
'calling weather API'
```

```
,
params
=
params
)
as
span
:
r
=
await
ctx
.
deps
.
client
.
get
(
'https://api.tomorrow.io/v4/weather/realtime'
,
params
=
params
)
r
.
raise_for_status
()
data
=
r
.
json
()
span
.
set_attribute
(
'response'
,
data
)
values
=
data
[
'data'
][
'values'
]
# https://docs.tomorrow.io/reference/data-layers-weather-codes
code_lookup
=
{
1000
:
'Clear, Sunny'
,
1100
:
'Mostly Clear'
,
1101
:
'Partly Cloudy'
,
1102
:
'Mostly Cloudy'
,
1001
:
'Cloudy'
,
2000
:
```

```
'Fog'
,
2100
:
'Light Fog'
,
4000
:
'Drizzle'
,
4001
:
'Rain'
,
4200
:
'Light Rain'
,
4201
:
'Heavy Rain'
,
5000
:
'Snow'
,
5001
:
'Flurries'
,
5100
:
'Light Snow'
,
5101
:
'Heavy Snow'
,
6000
:
'Freezing Drizzle'
,
6001
:
'Freezing Rain'
,
6200
:
'Light Freezing Rain'
,
6201
:
'Heavy Freezing Rain'
,
7000
:
'Ice Pellets'
,
7101
:
'Heavy Ice Pellets'
,
7102
:
'Light Ice Pellets'
,
8000
:
'Thunderstorm'
,
}
return
{
'temperature'
:
f
'
```

```
{
values
[
"temperatureApparent"
]
:
0.0f
}
°C'
,
'description'
:
code_lookup
.
get
(
values
[
'weatherCode'
],
'Unknown'
),
}
async
def
main
():
async
with
AsyncClient
()
as
client
:
# create a free API key at https://www.tomorrow.io/weather-api/
weather_api_key
=
os
.
getenv
(
'WEATHER_API_KEY'
)
# create a free API key at https://geocode.maps.co/
geo_api_key
=
os
.
getenv
(
'GEO_API_KEY'
)
deps
=
Deps
(
client
=
client
,
weather_api_key
=
weather_api_key
,
geo_api_key
=
geo_api_key
)
result
=
await
weather_agent
.
run
(
'What is the weather like in London and in Wiltshire?'
,
```

deps
=
deps
)
debug
(
result
)
print
(
'Response:'
,
result
.
data
)
if
__name__
==
'__main__'
:
asyncio
.
run
(
main
())

==
Page: RAG - PydanticAI
URL: https://ai.pydantic.dev/examples/rag/
==

RAG - PydanticAI
Skip to content
PydanticAI
RAG
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai
Introduction
Installation & Setup
Getting Help
Contributing
Documentation
Documentation
Agents
Dependencies
Results
Messages and chat history
Testing and Evals
Debugging and Monitoring
Examples
Examples
Pydantic Model
Weather agent
Bank support
SQL Generation
RAG
RAG
Table of contents
Example Code
Stream markdown
Stream whales
Chat App with FastAPI
API Reference
API Reference
pydantic_ai.Agent
pydantic_ai.tools
pydantic_ai.result
pydantic_ai.messages
pydantic_ai.exceptions
pydantic_ai.models.anthropic
pydantic_ai.models

pydantic_ai.models.openai
pydantic_ai.models.ollama
pydantic_ai.models.gemini
pydantic_ai.models.vertexai
pydantic_ai.models.groq
pydantic_ai.models.test
pydantic_ai.models.function
Table of contents
Example Code
Introduction
Examples
RAG
RAG search example. This demo allows you to ask question of the
logfire
documentation.
Demonstrates:
tools
agent dependencies
RAG search
This is done by creating a database containing each section of the markdown documentation, then registering
the search tool with the PydanticAI agent.
Logic for extracting sections from markdown files and a JSON file with that data is available in this gist
.
PostgreSQL with pgvector
is used as the search database, the easiest way to download and run pgvector is using Docker:

```
mkdir
postgres-data
docker
run
--rm
\
-e
POSTGRES_PASSWORD
=
postgres
\
-p
54320
:5432
\
-v
`
pwd
`
/postgres-data:/var/lib/postgresql/data
\
pgvector/pgvector:pg17
```

As with the
SQL gen
example, we run postgres on port
54320
to avoid conflicts with any other postgres instances you may have running.
We also mount the PostgreSQL
data
directory locally to persist the data if you need to stop and restart the container.
With that running and
dependencies installed and environment variables set
, we can build the search database with (
WARNING
: this requires the
OPENAI_API_KEY
env variable and will calling the OpenAI embedding API around 300 times to generate embeddings for each section of the documentation):

```
pip
uv
python
-m
pydantic_ai_examples.rag
build
uv
run
-m
pydantic_ai_examples.rag
build
```

(Note building the database doesn't use PydanticAI right now, instead it uses the OpenAI SDK

directly.)
You can then ask the agent a question with:
pip
uv
```
python
-m
pydantic_ai_examples.rag
search
"How do I configure logfire to work with FastAPI?"
```
```
uv
run
-m
pydantic_ai_examples.rag
search
"How do I configure logfire to work with FastAPI?"
```
Example Code
rag.py
```
from
__future__
import
annotations
as
_annotations
import
asyncio
import
re
import
sys
import
unicodedata
from
contextlib
import
asynccontextmanager
from
dataclasses
import
dataclass
import
asyncpg
import
httpx
import
logfire
import
pydantic_core
from
openai
import
AsyncOpenAI
from
pydantic
import
TypeAdapter
from
typing_extensions
import
AsyncGenerator
from
pydantic_ai
import
RunContext
from
pydantic_ai.agent
import
Agent
# 'if-token-present' means nothing will be sent (and the example will work) if you don't have
logfire configured
logfire
.
configure
(
send_to_logfire
=
'if-token-present'
)
```

```
logfire
.
instrument_asyncpg
()
@dataclass
class
Deps
:
openai
:
AsyncOpenAI
pool
:
asyncpg
.
Pool
agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
Deps
)
@agent
.
tool
async
def
retrieve
(
context
:
RunContext
[
Deps
],
search_query
:
str
)
->
str
:
"""Retrieve documentation sections based on a search query.
Args:
context: The call context.
search_query: The search query.
"""
with
logfire
.
span
(
'create embedding for {search_query=}'
,
search_query
=
search_query
):
embedding
=
await
context
.
deps
.
openai
.
embeddings
.
create
(
input
=
```

```
search_query
,
model
=
'text-embedding-3-small'
,
)
assert
(
len
(
embedding
.
data
)
==
1
),
f
'Expected 1 embedding, got
{
len
(
embedding
.
data
)
}
, doc query:
{
search_query
!r}
'
embedding
=
embedding
.
data
[
0
]
.
embedding
embedding_json
=
pydantic_core
.
to_json
(
embedding
)
.
decode
()
rows
=
await
context
.
deps
.
pool
.
fetch
(
'SELECT url, title, content FROM doc_sections ORDER BY embedding <-> $1 LIMIT 8'
,
embedding_json
,
)
return
'
\n\n
'
.
join
(
```

```
f
'#
{
row
[
"title"
]
}
\n
Documentation URL:
{
row
[
"url"
]
}
\n\n
{
row
[
"content"
]
}
\n
'
for
row
in
rows
)
async
def
run_agent
(
question
:
str
):
"""Entry point to run the agent and perform RAG based question answering."""
openai
=
AsyncOpenAI
()
logfire
.
instrument_openai
(
openai
)
logfire
.
info
(
'Asking "
{question}
"'
,
question
=
question
)
async
with
database_connect
(
False
)
as
pool
:
deps
=
Deps
(
openai
=
openai
```

```
,
pool
=
pool
)
answer
=
await
agent
.
run
(
question
,
deps
=
deps
)
print
(
answer
.
data
)
#######################################################
# The rest of this file is dedicated to preparing the #
# search database, and some utilities.                #
#######################################################
# JSON document from
# https://gist.github.com/samuelcolvin/4b5bb9bb163b1122ff17e29e48c10992
DOCS_JSON
=
(
'https://gist.githubusercontent.com/'
'samuelcolvin/4b5bb9bb163b1122ff17e29e48c10992/raw/'
'80c5925c42f1442c24963aaf5eb1a324d47afe95/logfire_docs.json'
)
async
def
build_search_db
():
"""Build the search database."""
async
with
httpx
.
AsyncClient
()
as
client
:
response
=
await
client
.
get
(
DOCS_JSON
)
response
.
raise_for_status
()
sections
=
sessions_ta
.
validate_json
(
response
.
content
)
openai
=
AsyncOpenAI
```

```
()
logfire
.
instrument_openai
(
openai
)
async
with
database_connect
(
True
)
as
pool
:
with
logfire
.
span
(
'create schema'
):
async
with
pool
.
acquire
()
as
conn
:
async
with
conn
.
transaction
():
await
conn
.
execute
(
DB_SCHEMA
)
sem
=
asyncio
.
Semaphore
(
10
)
async
with
asyncio
.
TaskGroup
()
as
tg
:
for
section
in
sections
:
tg
.
create_task
(
insert_doc_section
(
sem
,
openai
,
```

```
pool
,
section
))
async
def
insert_doc_section
(
sem
:
asyncio
.
Semaphore
,
openai
:
AsyncOpenAI
,
pool
:
asyncpg
.
Pool
,
section
:
DocsSection
,
)
->
None
:
async
with
sem
:
url
=
section
.
url
()
exists
=
await
pool
.
fetchval
(
'SELECT 1 FROM doc_sections WHERE url = $1'
,
url
)
if
exists
:
logfire
.
info
(
'Skipping {url=}'
,
url
=
url
)
with
logfire
.
span
(
'create embedding for {url=}'
,
url
=
url
):
```

```
embedding
=
await
openai
.
embeddings
.
create
(
input
=
section
.
embedding_content
(),
model
=
'text-embedding-3-small'
,
)
assert
(
len
(
embedding
.
data
)
==
1
),
f
'Expected 1 embedding, got
{
len
(
embedding
.
data
)
}
, doc section:
{
section
}
'
embedding
=
embedding
.
data
[
0
]
.
embedding
embedding_json
=
pydantic_core
.
to_json
(
embedding
)
.
decode
()
await
pool
.
execute
(
'INSERT INTO doc_sections (url, title, content, embedding) VALUES ($1, $2, $3, $4)'
,
url
,
section
```

```
.
title
,
section
.
content
,
embedding_json
,
)
@dataclass
class
DocsSection
:
id
:
int
parent
:
int
|
None
path
:
str
level
:
int
title
:
str
content
:
str
def
url
(
self
)
->
str
:
url_path
=
re
.
sub
(
r
'\.md$'
,
''
,
self
.
path
)
return
(
f
'https://logfire.pydantic.dev/docs/
{
url_path
}
/#
{
slugify
(
self
.
title
,
"-"
)
}
'
)
```

```
def
embedding_content
(
self
)
->
str
:
return
'
\n\n
'
.
join
((
f
'path:
{
self
.
path
}
'
,
f
'title:
{
self
.
title
}
'
,
self
.
content
))
sessions_ta
=
TypeAdapter
(
list
[
DocsSection
])
# pyright: reportUnknownMemberType=false
# pyright: reportUnknownVariableType=false
@asynccontextmanager
async
def
database_connect
(
create_db
:
bool
=
False
,
)
->
AsyncGenerator
[
asyncpg
.
Pool
,
None
]:
server_dsn
,
database
=
(
'postgresql://postgres:postgres@localhost:54320'
,
'pydantic_ai_rag'
,
```

```
)
if
create_db
:
with
logfire
.
span
(
'check and create DB'
):
conn
=
await
asyncpg
.
connect
(
server_dsn
)
try
:
db_exists
=
await
conn
.
fetchval
(
'SELECT 1 FROM pg_database WHERE datname = $1'
,
database
)
if
not
db_exists
:
await
conn
.
execute
(
f
'CREATE DATABASE
{
database
}
'
)
finally
:
await
conn
.
close
()
pool
=
await
asyncpg
.
create_pool
(
f
'
{
server_dsn
}
/
{
database
}
'
)
try
:
yield
```

```
pool
finally
:
await
pool
.
close
()
DB_SCHEMA
=
"""
CREATE EXTENSION IF NOT EXISTS vector;
CREATE TABLE IF NOT EXISTS doc_sections (
id serial PRIMARY KEY,
url text NOT NULL UNIQUE,
title text NOT NULL,
content text NOT NULL,
-- text-embedding-3-small returns a vector of 1536 floats
embedding vector(1536) NOT NULL
);
CREATE INDEX IF NOT EXISTS idx_doc_sections_embedding ON doc_sections USING hnsw (embedding
vector_l2_ops);
"""
def
slugify
(
value
:
str
,
separator
:
str
,
unicode
:
bool
=
False
)
->
str
:
"""Slugify a string, to make it URL friendly."""
# Taken unchanged from https://github.com/Python-
Markdown/markdown/blob/3.7/markdown/extensions/toc.py#L38
if
not
unicode
:
# Replace Extended Latin characters with ASCII, i.e. `žlutý` => `zluty`
value
=
unicodedata
.
normalize
(
'NFKD'
,
value
)
value
=
value
.
encode
(
'ascii'
,
'ignore'
)
.
decode
(
'ascii'
)
value
```

```
=
re
.
sub
(
r
'[^\w\s-]'
,
''
,
value
)
.
strip
()
.
lower
()
return
re
.
sub
(
rf
'[
{
separator
}
\s]+'
,
separator
,
value
)
if
__name__
==
'__main__'
:
action
=
sys
.
argv
[
1
]
if
len
(
sys
.
argv
)
>
1
else
None
if
action
==
'build'
:
asyncio
.
run
(
build_search_db
())
elif
action
==
==
'search'
:
if
len
(
```

```
sys
.
argv
)
==
3
:
q
=
sys
.
argv
[
2
]
else
:
q
=
'How do I configure logfire to work with FastAPI?'
asyncio
.
run
(
run_agent
(
q
))
else
:
print
(
'uv run --extra examples -m pydantic_ai_examples.rag build|search'
,
file
=
sys
.
stderr
,
)
sys
.
exit
(
1
)
```


==
Page: pydantic_ai.messages - PydanticAI
URL: https://ai.pydantic.dev/api/messages/
==

pydantic_ai.messages - PydanticAI
Skip to content
PydanticAI
pydantic_ai.messages
Initializing search
pydantic/pydantic-ai
PydanticAI
pydantic/pydantic-ai

timestamp
role
RetryPrompt
content
tool_name
tool_call_id
timestamp
role
ModelAnyResponse
ModelTextResponse
content
timestamp
role
ModelStructuredResponse
calls
timestamp
role
ToolCall
tool_name
args
tool_call_id
ArgsJson
args_json
MessagesTypeAdapter
Introduction
API Reference
pydantic_ai.messages
Message
module-attribute
Message
=
Union
[
SystemPrompt
,
UserPrompt
,
ToolReturn
,
RetryPrompt
,
ModelTextResponse
,
ModelStructuredResponse
,
]
Any message send to or returned by a model.
SystemPrompt
dataclass
A system prompt, generally written by the application developer.
This gives the model context and guidance on how to respond.
Source code in
pydantic_ai_slim/pydantic_ai/messages.py

```
@dataclass
class
SystemPrompt
:
"""A system prompt, generally written by the application developer.
This gives the model context and guidance on how to respond.
"""
content
:
str
"""The content of the prompt."""
role
:
```

Literal
[
'system'
]
=
'system'
"""Message type identifier, this type is available on all message as a discriminator."""
content
instance-attribute
content
:
str
The content of the prompt.
role
class-attribute
instance-attribute
role
:
Literal
[
'system'
]
=
'system'
Message type identifier, this type is available on all message as a discriminator.
UserPrompt
dataclass
A user prompt, generally written by the end user.
Content comes from the
user_prompt
parameter of
Agent.run
,
Agent.run_sync
, and
Agent.run_stream
.
Source code in
pydantic_ai_slim/pydantic_ai/messages.py

```
@dataclass
class
UserPrompt
:
"""A user prompt, generally written by the end user.
Content comes from the `user_prompt` parameter of [`Agent.run`][pydantic_ai.Agent.run],
[`Agent.run_sync`][pydantic_ai.Agent.run_sync], and [`Agent.run_stream`]
[pydantic_ai.Agent.run_stream].
"""
content
:
str
"""The content of the prompt."""
timestamp
:
datetime
=
field
(
default_factory
=
_now_utc
)
"""The timestamp of the prompt."""
```

role
:
Literal
[
'user'
]
=
'user'
"""Message type identifier, this type is available on all message as a discriminator."""
content
instance-attribute
content
:
str
The content of the prompt.
timestamp
class-attribute
instance-attribute
timestamp
:
datetime
=
field
(
default_factory
=
now_utc
)
The timestamp of the prompt.
role
class-attribute
instance-attribute
role
:
Literal
[
'user'
]
=
'user'
Message type identifier, this type is available on all message as a discriminator.
ToolReturn
dataclass
A tool return message, this encodes the result of running a tool.
Source code in
pydantic_ai_slim/pydantic_ai/messages.py
@dataclass
class
ToolReturn
:

```
"""A tool return message, this encodes the result of running a tool."""
tool_name
:
str
"""The name of the "tool" was called."""
content
:
Any
"""The return value."""
tool_call_id
:
str
|
None
=
None
"""Optional tool call identifier, this is used by some models including OpenAI."""
timestamp
:
datetime
=
field
(
default_factory
=
_now_utc
)
"""The timestamp, when the tool returned."""
role
:
Literal
[
'tool-return'
]
=
'tool-return'
"""Message type identifier, this type is available on all message as a discriminator."""
def
model_response_str
(
self
)
->
str
:
if
isinstance
(
self
.
content
,
str
):
return
self
.
content
else
:
return
tool_return_ta
.
dump_json
(
self
.
content
)
.
decode
()
def
model_response_object
(
self
)
```

```
->
dict
[
str
,
Any
]:
# gemini supports JSON dict return values, but no other JSON types, hence we wrap anything else in a
dict
if
isinstance
(
self
.
content
,
dict
):
return
tool_return_ta
.
dump_python
(
self
.
content
,
mode
=
'json'
)
# pyright: ignore[reportUnknownMemberType]
else
:
return
{
'return_value'
:
tool_return_ta
.
dump_python
(
self
.
content
,
mode
=
'json'
)}
```
tool_name
instance-attribute
tool_name
:
str
The name of the "tool" was called.
content
instance-attribute
content
:
Any
The return value.
tool_call_id
class-attribute
instance-attribute
tool_call_id
:
str
|
None
=
None
Optional tool call identifier, this is used by some models including OpenAI.
timestamp
class-attribute
instance-attribute
timestamp

:
datetime
=
field
(
default_factory
=
now_utc
)
The timestamp, when the tool returned.
role
class-attribute
instance-attribute
role
:
Literal
[
'tool-return'
]
=
'tool-return'
Message type identifier, this type is available on all message as a discriminator.
RetryPrompt
dataclass
A message back to a model asking it to try again.
This can be sent for a number of reasons:
Pydantic validation of tool arguments failed, here content is derived from a Pydantic
ValidationError
a tool raised a
ModelRetry
exception
no tool was found for the tool name
the model returned plain text when a structured response was expected
Pydantic validation of a structured response failed, here content is derived from a Pydantic
ValidationError
a result validator raised a
ModelRetry
exception
Source code in
pydantic_ai_slim/pydantic_ai/messages.py

```
@dataclass
class
RetryPrompt
:
"""A message back to a model asking it to try again.
This can be sent for a number of reasons:
* Pydantic validation of tool arguments failed, here content is derived from a Pydantic
[`ValidationError`][pydantic_core.ValidationError]
* a tool raised a [`ModelRetry`][pydantic_ai.exceptions.ModelRetry] exception
* no tool was found for the tool name
* the model returned plain text when a structured response was expected
* Pydantic validation of a structured response failed, here content is derived from a Pydantic
[`ValidationError`][pydantic_core.ValidationError]
* a result validator raised a [`ModelRetry`][pydantic_ai.exceptions.ModelRetry] exception
"""
content
:
list
[
pydantic_core
.
ErrorDetails
]
|
str
"""Details of why and how the model should retry.
If the retry was triggered by a [`ValidationError`][pydantic_core.ValidationError], this will be a
list of
error details.
"""
tool_name
:
str
|
None
=
None
"""The name of the tool that was called, if any."""
tool_call_id
:
str
|
None
=
None
"""Optional tool call identifier, this is used by some models including OpenAI."""
timestamp
:
datetime
=
field
(
default_factory
=
_now_utc
)
"""The timestamp, when the retry was triggered."""
role
:
Literal
[
'retry-prompt'
]
=
'retry-prompt'
"""Message type identifier, this type is available on all message as a discriminator."""
def
model_response
(
self
)
->
str
:
if
isinstance
```

```
(
self
.
content
,
str
):
description
=
self
.
content
else
:
json_errors
=
ErrorDetailsTa
.
dump_json
(
self
.
content
,
exclude
=
{
'__all__'
:
{
'ctx'
}},
indent
=
2
)
description
=
f
'
{
len
(
self
.
content
)
}
validation errors:
{
json_errors
.
decode
()
}
'
return
f
'
{
description
}
\n\n
Fix the errors and try again.'
content
instance-attribute
content
:
list
[
ErrorDetails
]
|
str
Details of why and how the model should retry.
If the retry was triggered by a
ValidationError
```

, this will be a list of
error details.
tool_name
class-attribute
instance-attribute
tool_name
:
str
|
None
=
None
The name of the tool that was called, if any.
tool_call_id
class-attribute
instance-attribute
tool_call_id
:
str
|
None
=
None
Optional tool call identifier, this is used by some models including OpenAI.
timestamp
class-attribute
instance-attribute
timestamp
:
datetime
=
field
(
default_factory
=
now_utc
)
The timestamp, when the retry was triggered.
role
class-attribute
instance-attribute
role
:
Literal
[
'retry-prompt'
]
=
'retry-prompt'
Message type identifier, this type is available on all message as a discriminator.
ModelAnyResponse
module-attribute
ModelAnyResponse
=
Union
[
ModelTextResponse
,
ModelStructuredResponse
]
Any response from a model.
ModelTextResponse
dataclass
A plain text response from a model.
Source code in
pydantic_ai_slim/pydantic_ai/messages.py

@dataclass
class
ModelTextResponse
:
"""A plain text response from a model."""
content
:
str
"""The text content of the response."""
timestamp
:
datetime
=
field
(
default_factory
=
_now_utc
)
"""The timestamp of the response.
If the model provides a timestamp in the response (as OpenAI does) that will be used.
"""
role
:
Literal
[
'model-text-response'
]
=
'model-text-response'
"""Message type identifier, this type is available on all message as a discriminator."""
content
instance-attribute
content
:
str
The text content of the response.
timestamp
class-attribute
instance-attribute
timestamp
:
datetime
=
field
(
default_factory
=
now_utc
)
The timestamp of the response.
If the model provides a timestamp in the response (as OpenAI does) that will be used.
role
class-attribute
instance-attribute
role
:
Literal
[
"model-text-response"
]
=
"model-text-response"
Message type identifier, this type is available on all message as a discriminator.
ModelStructuredResponse
dataclass
A structured response from a model.
This is used either to call a tool or to return a structured response from an agent run.
Source code in
pydantic_ai_slim/pydantic_ai/messages.py

```
@dataclass
class
ModelStructuredResponse
:
"""A structured response from a model.
This is used either to call a tool or to return a structured response from an agent run.
"""
calls
:
list
[
ToolCall
]
"""The tool calls being made."""
timestamp
:
datetime
=
field
(
default_factory
=
_now_utc
)
"""The timestamp of the response.
If the model provides a timestamp in the response (as OpenAI does) that will be used.
"""
role
:
Literal
[
'model-structured-response'
]
=
'model-structured-response'
"""Message type identifier, this type is available on all message as a discriminator."""
```
calls
instance-attribute
calls
:
list
[
ToolCall
]
The tool calls being made.
timestamp
class-attribute
instance-attribute
timestamp
:
datetime
=
field
(
default_factory
=
now_utc
)
The timestamp of the response.
If the model provides a timestamp in the response (as OpenAI does) that will be used.
role
class-attribute
instance-attribute
role
:
Literal

```
[
"model-structured-response"
]
=
(
"model-structured-response"
)
```
Message type identifier, this type is available on all message as a discriminator.

ToolCall

dataclass

A tool call from the agent.

Source code in
pydantic_ai_slim/pydantic_ai/messages.py

```
@dataclass
class
ToolCall
:
"""A tool call from the agent."""
tool_name
:
str
"""The name of the tool to call."""
args
:
ArgsJson
|
ArgsDict
"""The arguments to pass to the tool.
Either as JSON or a Python dictionary depending on how data was returned.
"""
tool_call_id
:
str
|
None
=
None
"""Optional tool call identifier, this is used by some models including OpenAI."""
@classmethod
def
from_json
(
cls
,
tool_name
:
str
,
args_json
:
```

```
str
,
tool_call_id
:
str
|
None
=
None
)
->
ToolCall
:
return
cls
(
tool_name
,
ArgsJson
(
args_json
),
tool_call_id
)
@classmethod
def
from_dict
(
cls
,
tool_name
:
str
,
args_dict
:
dict
[
str
,
Any
],
tool_call_id
:
str
|
None
=
None
)
->
ToolCall
:
return
cls
(
tool_name
,
ArgsDict
(
args_dict
),
tool_call_id
)
def
has_content
(
self
)
->
bool
:
if
isinstance
(
self
.
```

args
,
ArgsDict
):
return
any
(
self
.
args
.
args_dict
.
values
())
else
:
return
bool
(
self
.
args
.
args_json
)
tool_name
instance-attribute
tool_name
:
str
The name of the tool to call.
args
instance-attribute
args
:
ArgsJson
|
ArgsDict
The arguments to pass to the tool.
Either as JSON or a Python dictionary depending on how data was returned.
tool_call_id
class-attribute
instance-attribute
tool_call_id
:
str
|
None
=
None
Optional tool call identifier, this is used by some models including OpenAI.
ArgsJson
dataclass
Tool arguments as a JSON string.
Source code in
pydantic_ai_slim/pydantic_ai/messages.py
@dataclass
class
ArgsJson
:
"""Tool arguments as a JSON string."""
args_json
:
str
"""A JSON string of arguments."""
args_json
instance-attribute
args_json
:
str

A JSON string of arguments.
MessagesTypeAdapter
module-attribute
MessagesTypeAdapter
=
LazyTypeAdapter
(
list
[
Annotated
[
Message
,
Field
(
discriminator
=
"role"
)]]
)
Pydantic
TypeAdapter
for (de)serializing messages.

==
Page: pydantic_ai.models.gemini - PydanticAI
URL: https://ai.pydantic.dev/api/models/gemini/
==

pydantic_ai.models.gemini - PydanticAI

Table of contents

Introduction
API Reference
pydantic_ai.models.gemini
Custom interface to the
generativelanguage.googleapis.com
API using
HTTPX
and [Pydantic](https://docs.pydantic.dev/latest/.
The Google SDK for interacting with the
generativelanguage.googleapis.com
API
google-generativeai
reads like it was written by a
Java developer who thought they knew everything about OOP, spent 30 minutes trying to learn Python,
gave up and decided to build the library to prove how horrible Python is. It also doesn't use httpx
for HTTP requests,
and tries to implement tool calling itself, but doesn't use Pydantic or equivalent for validation.
We therefore implement support for the API directly.
Despite these shortcomings, the Gemini model is actually quite powerful and very fast.
Setup
For details on how to set up authentication with this model, see
model configuration for Gemini
.
GeminiModelName
module-attribute
GeminiModelName
=
Literal
[
"gemini-1.5-flash"
,
"gemini-1.5-flash-8b"
,
"gemini-1.5-pro"
,
"gemini-1.0-pro"
,
]
Named Gemini models.
See
the Gemini API docs
for a full list.
GeminiModel
dataclass
Bases:
Model
A model that uses Gemini via
generativelanguage.googleapis.com
API.
This is implemented from scratch rather than using a dedicated SDK, good API documentation is
available
here

.
Apart from
`__init__`
, all methods are private or match those of the base class.
Source code in
pydantic_ai_slim/pydantic_ai/models/gemini.py

```
@dataclass(init=False)
class GeminiModel
```

```
(
Model
):
"""A model that uses Gemini via `generativelanguage.googleapis.com` API.
This is implemented from scratch rather than using a dedicated SDK, good API documentation is
available [here](https://ai.google.dev/api).
Apart from `__init__`, all methods are private or match those of the base class.
"""
model_name
:
GeminiModelName
auth
:
AuthProtocol
http_client
:
AsyncHTTPClient
url
:
str
def
__init__
(
self
,
model_name
:
GeminiModelName
,
*
,
api_key
:
str
|
None
=
None
,
http_client
:
AsyncHTTPClient
|
None
=
None
,
url_template
:
str
=
'https://generativelanguage.googleapis.com/v1beta/models/
{model}
:'
,
):
"""Initialize a Gemini model.
Args:
model_name: The name of the model to use.
api_key: The API key to use for authentication, if not provided, the `GEMINI_API_KEY` environment
variable
will be used if available.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
url_template: The URL template to use for making requests, you shouldn't need to change this,
docs [here](https://ai.google.dev/gemini-api/docs/quickstart?lang=rest#make-first-request),
`model` is substituted with the model name, and `function` is added to the end of the URL.
"""
self
.
model_name
=
model_name
if
api_key
is
None
:
```

```
if
env_api_key
:=
os
.
getenv
(
'GEMINI_API_KEY'
):
api_key
=
env_api_key
else
:
raise
exceptions
.
UserError
(
'API key must be provided or set in the GEMINI_API_KEY environment variable'
)
self
.
auth
=
ApiKeyAuth
(
api_key
)
self
.
http_client
=
http_client
or
cached_async_http_client
()
self
.
url
=
url_template
.
format
(
model
=
model_name
)
async
def
agent_model
(
self
,
*
,
function_tools
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
],
)
->
GeminiAgentModel
:
```

```
return
GeminiAgentModel
(
http_client
=
self
.
http_client
,
model_name
=
self
.
model_name
,
auth
=
self
.
auth
,
url
=
self
.
url
,
function_tools
=
function_tools
,
allow_text_result
=
allow_text_result
,
result_tools
=
result_tools
,
)
def
name
(
self
)
->
str
:
return
self
.
model_name
__init__
__init__
(
model_name
:
GeminiModelName
,
*
,
api_key
:
str
|
None
=
None
,
http_client
:
AsyncClient
|
None
=
None
,
```

url_template
:
str
=
"https://generativelanguage.googleapis.com/v1beta/models/
{model}
:"
)
Initialize a Gemini model.
Parameters:
Name
Type
Description
Default
model_name
GeminiModelName
The name of the model to use.
required
api_key
str
| None
The API key to use for authentication, if not provided, the
GEMINI_API_KEY
environment variable
will be used if available.
None
http_client
AsyncClient
| None
An existing
httpx.AsyncClient
to use for making HTTP requests.
None
url_template
str
The URL template to use for making requests, you shouldn't need to change this,
docs
here
,
model
is substituted with the model name, and
function
is added to the end of the URL.
'https://generativelanguage.googleapis.com/v1beta/models/{model}:'
Source code in
pydantic_ai_slim/pydantic_ai/models/gemini.py
def
__init__
(

```
self
,
model_name
:
GeminiModelName
,
*
,
api_key
:
str
|
None
=
None
,
http_client
:
AsyncHTTPClient
|
None
=
None
,
url_template
:
str
=
'https://generativelanguage.googleapis.com/v1beta/models/
{model}
:'
,
):
"""Initialize a Gemini model.
Args:
model_name: The name of the model to use.
api_key: The API key to use for authentication, if not provided, the `GEMINI_API_KEY` environment
variable
will be used if available.
http_client: An existing `httpx.AsyncClient` to use for making HTTP requests.
url_template: The URL template to use for making requests, you shouldn't need to change this,
docs [here](https://ai.google.dev/gemini-api/docs/quickstart?lang=rest#make-first-request),
`model` is substituted with the model name, and `function` is added to the end of the URL.
"""
self
.
model_name
=
model_name
if
api_key
is
None
:
if
env_api_key
:=
os
.
getenv
(
'GEMINI_API_KEY'
):
api_key
=
env_api_key
else
:
raise
exceptions
.
UserError
(
'API key must be provided or set in the GEMINI_API_KEY environment variable'
)
self
.
```

```
auth = ApiKeyAuth(api_key)
self.http_client = http_client or cached_async_http_client()
self.url = url_template.format(model=model_name)
```

AuthProtocol

Bases: Protocol

Abstract definition for Gemini authentication.

Source code in pydantic_ai_slim/pydantic_ai/models/gemini.py

```
class AuthProtocol(Protocol):
    """Abstract definition for Gemini authentication."""

    async def headers(self) -> dict[str, str]: ...
```

ApiKeyAuth

dataclass

Authentication using an API key for the X-Goog-Api-Key header.

Source code in pydantic_ai_slim/pydantic_ai/models/gemini.py

```
@dataclass
class ApiKeyAuth:
    """Authentication using an API key for the `X-Goog-Api-Key` header."""
```

```
api_key
:
str
async
def
headers
(
self
)
->
dict
[
str
,
str
]:
# https://cloud.google.com/docs/authentication/api-keys-use#using-with-rest
return
{
'X-Goog-Api-Key'
:
self
.
api_key
}
```

GeminiAgentModel

dataclass

Bases:
AgentModel

Implementation of
AgentModel
for Gemini models.

Source code in
pydantic_ai_slim/pydantic_ai/models/gemini.py

```python
@dataclass(init=False)
class GeminiAgentModel(AgentModel):
    """Implementation of `AgentModel` for Gemini models."""

    http_client: AsyncHTTPClient
    model_name: GeminiModelName
    auth: AuthProtocol
    tools: _GeminiTools | None
    tool_config: _GeminiToolConfig | None
    url: str

    def __init__(
        self,
        http_client: AsyncHTTPClient,
        model_name: GeminiModelName,
        auth: AuthProtocol,
        url: str,
        function_tools
```

```
:
list
[
ToolDefinition
],
allow_text_result
:
bool
,
result_tools
:
list
[
ToolDefinition
],
):
check_allow_model_requests
()
tools
=
[
_function_from_abstract_tool
(
t
)
for
t
in
function_tools
]
if
result_tools
:
tools
+=
[
_function_from_abstract_tool
(
t
)
for
t
in
result_tools
]
if
allow_text_result
:
tool_config
=
None
else
:
tool_config
=
_tool_config
([
t
[
'name'
]
for
t
in
tools
])
self
.
http_client
=
http_client
self
.
model_name
=
model_name
self
```

```
.
auth
=
auth
self
.
tools
=
_GeminiTools
(
function_declarations
=
tools
)
if
tools
else
None
self
.
tool_config
=
tool_config
self
.
url
=
url
async
def
request
(
self
,
messages
:
list
[
Message
])
->
tuple
[
ModelAnyResponse
,
result
.
Cost
]:
async
with
self
.
_make_request
(
messages
,
False
)
as
http_response
:
response
=
_gemini_response_ta
.
validate_json
(
await
http_response
.
aread
())
return
self
.
_process_response
```

```
(
response
),
_metadata_as_cost
(
response
)
@asynccontextmanager
async
def
request_stream
(
self
,
messages
:
list
[
Message
])
->
AsyncIterator
[
EitherStreamedResponse
]:
async
with
self
.
_make_request
(
messages
,
True
)
as
http_response
:
yield
await
self
.
_process_streamed_response
(
http_response
)
@asynccontextmanager
async
def
_make_request
(
self
,
messages
:
list
[
Message
],
streamed
:
bool
)
->
AsyncIterator
[
HTTPResponse
]:
contents
:
list
[
_GeminiContent
]
=
[]
sys_prompt_parts
```

```
:
list
[
_GeminiTextPart
]
=
[]
for
m
in
messages
:
either_content
=
self
.
_message_to_gemini
(
m
)
if
left
:=
either_content
.
left
:
sys_prompt_parts
.
append
(
left
.
value
)
else
:
contents
.
append
(
either_content
.
right
)
request_data
=
_GeminiRequest
(
contents
=
contents
)
if
sys_prompt_parts
:
request_data
[
'system_instruction'
]
=
_GeminiTextContent
(
role
=
'user'
,
parts
=
sys_prompt_parts
)
if
self
.
tools
is
not
```

```
None
:
request_data
[
'tools'
]
=
self
.
tools
if
self
.
tool_config
is
not
None
:
request_data
[
'tool_config'
]
=
self
.
tool_config
url
=
self
.
url
+
(
'streamGenerateContent'
if
streamed
else
'generateContent'
)
headers
=
{
'Content-Type'
:
'application/json'
,
'User-Agent'
:
get_user_agent
(),
**
await
self
.
auth
.
headers
(),
}
request_json
=
_gemini_request_ta
.
dump_json
(
request_data
,
by_alias
=
True
)
async
with
self
.
http_client
.
```

```
stream
(
'POST'
,
url
,
content
=
request_json
,
headers
=
headers
)
as
r
:
if
r
.
status_code
!=
200
:
await
r
.
aread
()
raise
exceptions
.
UnexpectedModelBehavior
(
f
'Unexpected response from gemini
{
r
.
status_code
}
'
,
r
.
text
)
yield
r
@staticmethod
def
_process_response
(
response
:
_GeminiResponse
)
->
ModelAnyResponse
:
either
=
_extract_response_parts
(
response
)
if
left
:=
either
.
left
:
return
_structured_response_from_parts
(
left
```

```
.
value
)
else
:
return
ModelTextResponse
(
content
=
''
.
join
(
part
[
'text'
]
for
part
in
either
.
right
))
@staticmethod
async
def
_process_streamed_response
(
http_response
:
HTTPResponse
)
->
EitherStreamedResponse
:
"""Process a streamed response, and prepare a streaming response to return."""
aiter_bytes
=
http_response
.
aiter_bytes
()
start_response
:
_GeminiResponse
|
None
=
None
content
=
bytearray
()
async
for
chunk
in
aiter_bytes
:
content
.
extend
(
chunk
)
responses
=
_gemini_streamed_response_ta
.
validate_json
(
content
,
experimental_allow_partial
=
```

```
'trailing-strings'
,
)
if
responses
:
last
=
responses
[
-
1
]
if
last
[
'candidates'
]
and
last
[
'candidates'
][
0
][
'content'
][
'parts'
]:
start_response
=
last
break
if
start_response
is
None
:
raise
UnexpectedModelBehavior
(
'Streamed response ended without content or tool calls'
)
if
_extract_response_parts
(
start_response
)
.
is_left
():
return
GeminiStreamStructuredResponse
(
_content
=
content
,
_stream
=
aiter_bytes
)
else
:
return
GeminiStreamTextResponse
(
_json_content
=
content
,
_stream
=
aiter_bytes
)
@staticmethod
def
```

```
_message_to_gemini
(
m
:
Message
)
->
_utils
.
Either
[
_GeminiTextPart
,
_GeminiContent
]:
"""Convert a message to a _GeminiTextPart for "system_instructions" or _GeminiContent for
"contents"."""
if
m
.
role
==
'system'
:
# SystemPrompt ->
return
_utils
.
Either
(
left
=
_GeminiTextPart
(
text
=
m
.
content
))
elif
m
.
role
==
'user'
:
# UserPrompt ->
return
_utils
.
Either
(
right
=
_content_user_text
(
m
.
content
))
elif
m
.
role
==
'tool-return'
:
# ToolReturn ->
return
_utils
.
Either
(
right
=
_content_function_return
```

```
(
m
))
elif
m
.
role
==
'retry-prompt'
:
# RetryPrompt ->
return
_utils
.
Either
(
right
=
_content_function_retry
(
m
))
elif
m
.
role
==
'model-text-response'
:
# ModelTextResponse ->
return
_utils
.
Either
(
right
=
_content_model_text
(
m
.
content
))
elif
m
.
role
==
'model-structured-response'
:
# ModelStructuredResponse ->
return
_utils
.
Either
(
right
=
_content_function_call
(
m
))
else
:
assert_never
(
m
)
```

GeminiStreamTextResponse
dataclass
Bases:
StreamTextResponse
Implementation of
StreamTextResponse
for the Gemini model.
Source code in
pydantic_ai_slim/pydantic_ai/models/gemini.py

```
@dataclass
class
GeminiStreamTextResponse
(
StreamTextResponse
):
"""Implementation of `StreamTextResponse` for the Gemini model."""
_json_content
:
bytearray
_stream
:
AsyncIterator
[
bytes
]
_position
:
int
=
0
_timestamp
:
datetime
=
field
(
default_factory
=
_utils
.
now_utc
,
init
```

```
=
False
)
_cost
:
result
.
Cost
=
field
(
default_factory
=
result
.
Cost
,
init
=
False
)
async
def
__anext__
(
self
)
->
None
:
chunk
=
await
self
.
_stream
.
__anext__
()
self
.
_json_content
.
extend
(
chunk
)
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
Iterable
[
str
]:
if
final
:
all_items
=
pydantic_core
.
from_json
(
self
.
_json_content
```

```
)
new_items
=
all_items
[
self
.
_position
:]
self
.
_position
=
len
(
all_items
)
new_responses
=
_gemini_streamed_response_ta
.
validate_python
(
new_items
)
else
:
all_items
=
pydantic_core
.
from_json
(
self
.
_json_content
,
allow_partial
=
True
)
new_items
=
all_items
[
self
.
_position
:
-
1
]
self
.
_position
=
len
(
all_items
)
-
1
new_responses
=
_gemini_streamed_response_ta
.
validate_python
(
new_items
,
experimental_allow_partial
=
'trailing-strings'
)
for
r
in
```

```
new_responses
:
self
.
_cost
+=
_metadata_as_cost
(
r
)
parts
=
r
[
'candidates'
][
0
][
'content'
][
'parts'
]
if
_all_text_parts
(
parts
):
for
part
in
parts
:
yield
part
[
'text'
]
else
:
raise
UnexpectedModelBehavior
(
'Streamed response with unexpected content, expected all parts to be text'
)
def
cost
(
self
)
->
result
.
Cost
:
return
self
.
_cost
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp
```

GeminiStreamStructuredResponse

dataclass

Bases:
StreamStructuredResponse

Implementation of
StreamStructuredResponse
for the Gemini model.

Source code in
pydantic_ai_slim/pydantic_ai/models/gemini.py

```
@dataclass
class
GeminiStreamStructuredResponse
(
StreamStructuredResponse
):
"""Implementation of `StreamStructuredResponse` for the Gemini model."""
_content
:
bytearray
_stream
:
AsyncIterator
[
bytes
]
_timestamp
:
datetime
=
field
(
default_factory
=
_utils
.
now_utc
,
init
```

```
=
False
)
_cost
:
result
.
Cost
=
field
(
default_factory
=
result
.
Cost
,
init
=
False
)
async
def
__anext__
(
self
)
->
None
:
chunk
=
await
self
.
_stream
.
__anext__
()
self
.
_content
.
extend
(
chunk
)
def
get
(
self
,
*
,
final
:
bool
=
False
)
->
ModelStructuredResponse
:
"""Get the `ModelStructuredResponse` at this point.
NOTE: It's not clear how the stream of responses should be combined because Gemini seems to always
reply with a single response, when returning a structured data.
I'm therefore assuming that each part contains a complete tool call, and not trying to combine data
from
separate parts.
"""
responses
=
_gemini_streamed_response_ta
.
validate_json
(
self
```

```
.
_content
,
experimental_allow_partial
=
'off'
if
final
else
'trailing-strings'
,
)
combined_parts
:
list
[
_GeminiFunctionCallPart
]
=
[]
self
.
_cost
=
result
.
Cost
()
for
r
in
responses
:
self
.
_cost
+=
_metadata_as_cost
(
r
)
candidate
=
r
[
'candidates'
][
0
]
parts
=
candidate
[
'content'
][
'parts'
]
if
_all_function_call_parts
(
parts
):
combined_parts
.
extend
(
parts
)
elif
not
candidate
.
get
(
'finish_reason'
):
# you can get an empty text part along with the finish_reason, so we ignore that case
```

raise
UnexpectedModelBehavior
(
'Streamed response with unexpected content, expected all parts to be function calls'
)
return
_structured_response_from_parts
(
combined_parts
,
timestamp
=
self
.
_timestamp
)
def
cost
(
self
)
->
result
.
Cost
:
return
self
.
_cost
def
timestamp
(
self
)
->
datetime
:
return
self
.
_timestamp
get
get
(
*
,
final
:
bool
=
False
)
->
ModelStructuredResponse
Get the
ModelStructuredResponse
at this point.
NOTE: It's not clear how the stream of responses should be combined because Gemini seems to always reply with a single response, when returning a structured data.
I'm therefore assuming that each part contains a complete tool call, and not trying to combine data from
separate parts.
Source code in
pydantic_ai_slim/pydantic_ai/models/gemini.py

```
def get(self, *, final: bool = False) -> ModelStructuredResponse:
    """Get the `ModelStructuredResponse` at this point.
    NOTE: It's not clear how the stream of responses should be combined because Gemini seems to always
    reply with a single response, when returning a structured data.
    I'm therefore assuming that each part contains a complete tool call, and not trying to combine data
    from
    separate parts.
    """
    responses = _gemini_streamed_response_ta.validate_json(
        self._content,
        experimental_allow_partial='off' if final else 'trailing-strings',
    )
    combined_parts: list[_GeminiFunctionCallPart] = []
    self._cost = result.Cost()
    for r in responses
```

```
:
self
.
_cost
+=
_metadata_as_cost
(
r
)
candidate
=
r
[
'candidates'
][
0
]
parts
=
candidate
[
'content'
][
'parts'
]
if
_all_function_call_parts
(
parts
):
combined_parts
.
extend
(
parts
)
elif
not
candidate
.
get
(
'finish_reason'
):
# you can get an empty text part along with the finish_reason, so we ignore that case
raise
UnexpectedModelBehavior
(
'Streamed response with unexpected content, expected all parts to be function calls'
)
return
_structured_response_from_parts
(
combined_parts
,
timestamp
=
self
.
_timestamp
)
```


==
Page: Dependencies - PydanticAI
URL: https://ai.pydantic.dev/dependencies/
==

Dependencies - PydanticAI

PydanticAI uses a dependency injection system to provide data and services to your agent's
system prompts
,
tools
and
result validators
.
Matching PydanticAI's design philosophy, our dependency system tries to use existing best practice in Python development rather than inventing esoteric "magic", this should make dependencies type-safe, understandable easier to test and ultimately easier to deploy in production.
Defining Dependencies
Dependencies can be any python type. While in simple cases you might be able to pass a single object as a dependency (e.g. an HTTP connection),
dataclasses
are generally a convenient container when your dependencies included multiple objects.
Here's an example of defining an agent that requires dependencies.
(
Note:
dependencies aren't actually used in this example, see

```
Accessing Dependencies
below)
unused_dependencies.py
from
dataclasses
import
dataclass
import
httpx
from
pydantic_ai
import
Agent
@dataclass
class
MyDeps
:
# (1)!
api_key
:
str
http_client
:
httpx
.
AsyncClient
agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
MyDeps
,
# (2)!
)
async
def
main
():
async
with
httpx
.
AsyncClient
()
as
client
:
deps
=
MyDeps
(
'foobar'
,
client
)
result
=
await
agent
.
run
(
'Tell me a joke.'
,
deps
=
deps
,
# (3)!
)
print
(
result
```

.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
Define a dataclass to hold dependencies.
Pass the dataclass type to the
deps_type
argument of the
Agent
constructor
.
Note
: we're passing the type here, NOT an instance, this parameter is not actually used at runtime, it's here so we can get full type checking of the agent.
When running the agent, pass an instance of the dataclass to the
deps
parameter.
(This example is complete, it can be run "as is")
Accessing Dependencies
Dependencies are accessed through the
RunContext
type, this should be the first parameter of system prompt functions etc.
system_prompt_dependencies.py

```
from
dataclasses
import
dataclass
import
httpx
from
pydantic_ai
import
Agent
,
RunContext
@dataclass
class
MyDeps
:
api_key
:
str
http_client
:
httpx
.
AsyncClient
agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
MyDeps
,
)
@agent
.
system_prompt
# (1)!
async
def
get_system_prompt
(
ctx
:
RunContext
[
MyDeps
])
->
str
:
# (2)!
response
```

```
=
await
ctx
.
deps
.
http_client
.
get
(
# (3)!
'https://example.com'
,
headers
=
{
'Authorization'
:
f
'Bearer
{
ctx
.
deps
.
api_key
}
'
},
# (4)!
)
response
.
raise_for_status
()
return
f
'Prompt:
{
response
.
text
}
'
async
def
main
():
async
with
httpx
.
AsyncClient
()
as
client
:
deps
=
MyDeps
(
'foobar'
,
client
)
result
=
await
agent
.
run
(
'Tell me a joke.'
,
deps
=
deps
```

)
print
(
result
.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
RunContext
may optionally be passed to a
system_prompt
function as the only argument.
RunContext
is parameterized with the type of the dependencies, if this type is incorrect, static type checkers will raise an error.
Access dependencies through the
.deps
attribute.
Access dependencies through the
.deps
attribute.
(This example is complete, it can be run "as is")
Asynchronous vs. Synchronous dependencies
System prompt functions
,
function tools
and
result validators
are all run in the async context of an agent run.
If these functions are not coroutines (e.g.
async def
) they are called with
run_in_executor
in a thread pool, it's therefore marginally preferable
to use
async
methods where dependencies perform IO, although synchronous dependencies should work fine too.
run
vs.
run_sync
and Asynchronous vs. Synchronous dependencies
Whether you use synchronous or asynchronous dependencies, is completely independent of whether you use
run
or
run_sync
—
run_sync
is just a wrapper around
run
and agents are always run in an async context.
Here's the same example as above, but with a synchronous dependency:
sync_dependencies.py
from
dataclasses
import
dataclass
import
httpx
from
pydantic_ai
import
Agent
,
RunContext
@dataclass
class
MyDeps
:
api_key
:
str
http_client
:
httpx
.
Client

```
# (1)!
agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
MyDeps
,
)
@agent
.
system_prompt
def
get_system_prompt
(
ctx
:
RunContext
[
MyDeps
])
->
str
:
# (2)!
response
=
ctx
.
deps
.
http_client
.
get
(
'https://example.com'
,
headers
=
{
'Authorization'
:
f
'Bearer
{
ctx
.
deps
.
api_key
}
'
}
)
response
.
raise_for_status
()
return
f
'Prompt:
{
response
.
text
}
'
async
def
main
():
deps
=
MyDeps
```

```
(
'foobar'
,
httpx
.
Client
())
result
=
await
agent
.
run
(
'Tell me a joke.'
,
deps
=
deps
,
)
print
(
result
.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
```

Here we use a synchronous
httpx.Client
instead of an asynchronous
httpx.AsyncClient
.
To match the synchronous dependency, the system prompt function is now a plain function, not a coroutine.
(This example is complete, it can be run "as is")
Full Example
As well as system prompts, dependencies can be used in
tools
and
result validators
.
full_example.py

```
from
dataclasses
import
dataclass
import
httpx
from
pydantic_ai
import
Agent
,
ModelRetry
,
RunContext
@dataclass
class
MyDeps
:
api_key
:
str
http_client
:
httpx
.
AsyncClient
agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
```

```
MyDeps
,
)
@agent
.
system_prompt
async
def
get_system_prompt
(
ctx
:
RunContext
[
MyDeps
])
->
str
:
response
=
await
ctx
.
deps
.
http_client
.
get
(
'https://example.com'
)
response
.
raise_for_status
()
return
f
'Prompt:
{
response
.
text
}
'
@agent
.
tool
# (1)!
async
def
get_joke_material
(
ctx
:
RunContext
[
MyDeps
],
subject
:
str
)
->
str
:
response
=
await
ctx
.
deps
.
http_client
.
get
(
```

```
'https://example.com#jokes'
,
params
=
{
'subject'
:
subject
},
headers
=
{
'Authorization'
:
f
'Bearer
{
ctx
.
deps
.
api_key
}
'
},
)
response
.
raise_for_status
()
return
response
.
text
@agent
.
result_validator
# (2)!
async
def
validate_result
(
ctx
:
RunContext
[
MyDeps
],
final_response
:
str
)
->
str
:
response
=
await
ctx
.
deps
.
http_client
.
post
(
'https://example.com#validate'
,
headers
=
{
'Authorization'
:
f
'Bearer
{
ctx
```

```
.
deps
.
api_key
}
'
},
params
=
{
'query'
:
final_response
},
)
if
response
.
status_code
==
400
:
raise
ModelRetry
(
f
'invalid response:
{
response
.
text
}
'
)
response
.
raise_for_status
()
return
final_response
async
def
main
():
async
with
httpx
.
AsyncClient
()
as
client
:
deps
=
MyDeps
(
'foobar'
,
client
)
result
=
await
agent
.
run
(
'Tell me a joke.'
,
deps
=
deps
)
print
(
result
```

.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
To pass
RunContext
to a tool, use the
tool
decorator.
RunContext
may optionally be passed to a
result_validator
function as the first argument.
(This example is complete, it can be run "as is")
Overriding Dependencies
When testing agents, it's useful to be able to customise dependencies.
While this can sometimes be done by calling the agent directly within unit tests, we can also
override dependencies
while calling application code which in turn calls the agent.
This is done via the
override
method on the agent.
joke_app.py
```
from
dataclasses
import
dataclass
import
httpx
from
pydantic_ai
import
Agent
,
RunContext
@dataclass
class
MyDeps
:
api_key
:
str
http_client
:
httpx
.
AsyncClient
async
def
system_prompt_factory
(
self
)
->
str
:
# (1)!
response
=
await
self
.
http_client
.
get
(
'https://example.com'
)
response
.
raise_for_status
()
return
f
'Prompt:
{
response
```

```
.
text
}
'
joke_agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
MyDeps
)
@joke_agent
.
system_prompt
async
def
get_system_prompt
(
ctx
:
RunContext
[
MyDeps
])
->
str
:
return
await
ctx
.
deps
.
system_prompt_factory
()
# (2)!
async
def
application_code
(
prompt
:
str
)
->
str
:
# (3)!
...
...
# now deep within application code we call our agent
async
with
httpx
.
AsyncClient
()
as
client
:
app_deps
=
MyDeps
(
'foobar'
,
client
)
result
=
await
joke_agent
.
run
```

(
prompt
,
deps
=
app_deps
)
(4)!
return
result
.
data
Define a method on the dependency to make the system prompt easier to customise.
Call the system prompt factory from within the system prompt function.
Application code that calls the agent, in a real application this might be an API endpoint.
Call the agent from within the application code, in a real application this call might be deep within a call stack. Note
app_deps
here will NOT be used when deps are overridden.
test_joke_app.py
from
joke_app
import
MyDeps
,
application_code
,
joke_agent
class
TestMyDeps
(
MyDeps
):
(1)!
async
def
system_prompt_factory
(
self
)
->
str
:
return
'test prompt'
async
def
test_application_code
():
test_deps
=
TestMyDeps
(
'test_key'
,
None
)
(2)!
with
joke_agent
.
override
(
deps
=
test_deps
):
(3)!
joke
=
await
application_code
(
'Tell me a joke.'
)
(4)!
assert

```
joke
.
startswith
(
'Did you hear about the toothpaste scandal?'
)
```
Define a subclass of
MyDeps
in tests to customise the system prompt factory.

Create an instance of the test dependency, we don't need to pass an
http_client
here as it's not used.

Override the dependencies of the agent for the duration of the
with
block,
test_deps
will be used when the agent is run.

Now we can safely call our application code, the agent will use the overridden dependencies.

Agents as dependencies of other Agents

Since dependencies can be any python type, and agents are just python objects, agents can be dependencies of other agents.

agents_as_dependencies.py
```
from
dataclasses
import
dataclass
from
pydantic_ai
import
Agent
,
RunContext
@dataclass
class
MyDeps
:
factory_agent
:
Agent
[
None
,
list
[
str
]]
joke_agent
=
Agent
(
'openai:gpt-4o'
,
deps_type
=
MyDeps
,
system_prompt
=
(
'Use the "joke_factory" to generate some jokes, then choose the best. '
'You must return just a single joke.'
),
)
factory_agent
=
Agent
(
'gemini-1.5-pro'
,
result_type
=
list
[
str
])
@joke_agent
.
```

```
tool
async
def
joke_factory
(
ctx
:
RunContext
[
MyDeps
],
count
:
int
)
->
str
:
r
=
await
ctx
.
deps
.
factory_agent
.
run
(
f
'Please generate
{
count
}
jokes.'
)
return
'
\n
'
.
join
(
r
.
data
)
result
=
joke_agent
.
run_sync
(
'Tell me a joke.'
,
deps
=
MyDeps
(
factory_agent
))
print
(
result
.
data
)
#> Did you hear about the toothpaste scandal? They called it Colgate.
```

Examples

The following examples demonstrate how to use dependencies in PydanticAI:

Weather Agent

SQL Generation

RAG

==

Page: Email Protection | Cloudflare
URL: https://ai.pydantic.dev/cdn-cgi/l/email-protection
==

Email Protection | Cloudflare
Please enable cookies.
Email Protection
You are unable to access this email address
pydantic.dev
The website from which you got to this page is protected by Cloudflare. Email addresses on that page have been hidden in order to keep them from being accessed by malicious bots.
You must enable Javascript in your browser in order to decode the e-mail address
.
If you have a website and are interested in protecting it in a similar way, you can
sign up for Cloudflare
.
How does Cloudflare protect email addresses on website from spammers?
Can I sign up for Cloudflare?
Cloudflare Ray ID:
8f0cbcb50c539718
•
Your IP:
Click to reveal
85.105.17.166
•
Performance & security by
Cloudflare