`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-3.1-70b-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",
    "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`

```python
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`

```python
@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`

```python
request(
    messages: list[Message],
) -> tuple[ModelAnyResponse, Cost]
```

Make a request to the model.

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

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

request_stream `async`

```python
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`

```python
@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`

```python
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`

```python
@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`

```python
@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`

```
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`

```python
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

```python
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`

```python
@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]
```