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`

```
110  def cost(self) -> Cost:
111      """Return the cost of the whole run."""
112      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`

```python
@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`

```
86  def new_messages(self) -> list[messages.Message]:
87      """Return new messages associated with this run.
88
89      System prompts and any messages from older runs are excluded.
90      """
91      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  def new_messages_json(self) -> bytes:
94      """Return new messages from [`new_messages`][..new_messages] as JSON bytes."""
95      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`

```
136 async def stream(self, *, debounce_by: float | None = 0.1) -> AsyncIterator[ResultData]:
137     """Stream the response as an async iterable.
138
139     The pydantic validator for structured data will be called in
140     [partial mode](https://docs.pydantic.dev/dev/concepts/experimental/#partial-validation)
141     on each iteration.
142
143     Args:
144         debounce_by: by how much (if at all) to debounce/group the response chunks by. `None` means no debouncing.
145             Debouncing is particularly important for long structured responses to reduce the overhead of
146             performing validation as each token is received.
147
148     Returns:
149         An async iterable of the response data.
150     """
151     if isinstance(self._stream_response, models.StreamTextResponse):
152         async for text in self.stream_text(debounce_by=debounce_by):
153             yield cast(ResultData, text)
154     else:
155         async for structured_message, is_last in self.stream_structured(debounce_by=debounce_by):
156             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`

```python
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`

```python
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`

```python
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`

```python
get_data() -> ResultData
```

Stream the whole response, validate and return it.

Source code in `pydantic_ai_slim/pydantic_ai/result.py`

```python
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`

```python
is_structured: bool
```

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

cost

```python
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`

```python
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

```python
timestamp() -> datetime
```

Get the timestamp of the response.

Source code in `pydantic_ai_slim/pydantic_ai/result.py`

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

validate_structured_result `async`

```python
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`

```python
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`

```python
@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`

```python
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)
```