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`

```python
@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:

    ```py
    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:
        ```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
        ```

        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:
        ```py
        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,
```

```python
    ) -> AsyncIterator[result.StreamedRunResult[AgentDeps, ResultData]]:
        """Run the agent with a user prompt in async mode, returning a streamed response.

        Example:
        ```py
        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
```

```python
            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:
        ```py
        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:
        ```py
        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:
        ```py
        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:
        ```py
        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."""
```

```python
        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))
```

```python
                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`

```python
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`

```python
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`

```python
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:

```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
```

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`

Name	Type	Description	Default
`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`

```python
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:
    ```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
    ```

    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

```python
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:

```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
```

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`

```python
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:
    ```py
    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`

```python
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:

```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
```

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`

```python
@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:
    ```py
    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`

```python
model: Model | KnownModelName | None
```

The default model configured for this agent.

override

```python
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`

```python
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

```python
system_prompt(
    func: Callable[[RunContext[AgentDeps]], str]
) -> Callable[[RunContext[AgentDeps]], str]
```

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

```python
system_prompt(func: Callable[[], str]) -> Callable[[], str]
```

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

```python
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:

```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)
```

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

```
452  def system_prompt(
453      self, func: _system_prompt.SystemPromptFunc[AgentDeps], /
454  ) -> _system_prompt.SystemPromptFunc[AgentDeps]:
455      """Decorator to register a system prompt function.
456
457      Optionally takes [`RunContext`][pydantic_ai.tools.RunContext] as its only argument.
458      Can decorate a sync or async functions.
459
460      Overloads for every possible signature of `system_prompt` are included so the decorator doesn't obscure
461      the type of the function, see `tests/typed_agent.py` for tests.
462
463      Example:
464      ```py
465      from pydantic_ai import Agent, RunContext
466
467      agent = Agent('test', deps_type=str)
468
469      @agent.system_prompt
470      def simple_system_prompt() -> str:
471          return 'foobar'
472
473      @agent.system_prompt
474      async def async_system_prompt(ctx: RunContext[str]) -> str:
475          return f'{ctx.deps} is the best'
476
477      result = agent.run_sync('foobar', deps='spam')
478      print(result.data)
479      #> success (no tool calls)
480      ```
481      """
482      self._system_prompt_functions.append(_system_prompt.SystemPromptRunner(func))
483      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`

```python
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:
    ```py
    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

```python
tool_plain(
    func: ToolFuncPlain[ToolParams],
) -> ToolFuncPlain[ToolParams]
```

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

```python
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:

```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}
```

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`

```python
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:
    ```py
    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

```python
result_validator(
    func: Callable[
        [RunContext[AgentDeps], ResultData], ResultData
    ]
) -> Callable[
    [RunContext[AgentDeps], ResultData], ResultData
]
```

```python
result_validator(
    func: Callable[
        [RunContext[AgentDeps], ResultData],
        Awaitable[ResultData],
    ]
) -> Callable[
    [RunContext[AgentDeps], ResultData],
    Awaitable[ResultData],
]
```

```python
result_validator(
    func: Callable[[ResultData], ResultData]
) -> Callable[[ResultData], ResultData]
```

```python
result_validator(
    func: Callable[[ResultData], Awaitable[ResultData]]
) -> Callable[[ResultData], Awaitable[ResultData]]
```

```python
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:

```py
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`

```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:
    ```py
    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
```