

## 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
```



```

44 @final
45 @dataclass(init=False)
46 class Agent(Generic[AgentDeps, ResultData]):
47     """Class for defining "agents" - a way to have a specific type of "conversation" with an LLM.
48
49     Agents are generic in the dependency type they take ['AgentDeps'][pydantic_ai.tools.AgentDeps]
50     and the result data type they return, ['ResultData'][pydantic_ai.result.ResultData].
51
52     By default, if neither generic parameter is customised, agents have type 'Agent[None, str]'.
53
54     Minimal usage example:
55
56     ```py
57     from pydantic_ai import Agent
58
59     agent = Agent('openai:gpt-4o')
60     result = agent.run_sync('What is the capital of France?')
61     print(result.data)
62     #> Paris
63     ```
64
65     # we use dataclass fields in order to conveniently know what attributes are available
66     model: models.Model | models.KnownModelName | None
67     """The default model configured for this agent."""
68
69     name: str | None
70     """The name of the agent, used for logging.
71
72     If 'None', we try to infer the agent name from the call frame when the agent is first run.
73
74     """
75
76     last_run_messages: list[_messages.Message] | None = None
77     """The messages from the last run, useful when a run raised an exception.
78
79     Note: these are not used by the agent, e.g. in future runs, they are just stored for developers' convenience.
80
81     """
82
83     _result_schema: _result.ResultSchema[ResultData] | None = field(repr=False)
84     _result_validators: list[_result.ResultValidator[AgentDeps, ResultData]] = field(repr=False)
85     _allow_text_result: bool = field(repr=False)
86     _system_prompts: tuple[str, ...] = field(repr=False)
87     _function_tools: dict[str, Tool[AgentDeps]] = field(repr=False)
88     _default_retries: int = field(repr=False)
89     _system_prompt_functions: list[_system_prompt.SystemPromptRunner[AgentDeps]] = field(repr=False)
90     _deps_type: type[AgentDeps] = field(repr=False)
91     _max_result_retries: int = field(repr=False)
92     _current_result_retry: int = field(repr=False)
93     _override_deps: _utils.Option[AgentDeps] = field(default=None, repr=False)
94     _override_model: _utils.Option[models.Model] = field(default=None, repr=False)
95
96     def __init__(
97         self,
98         model: models.Model | models.KnownModelName | None = None,
99         *,
100         result_type: type[ResultData] = str,
101         system_prompt: str | Sequence[str] = (),
102         deps_type: type[AgentDeps] = NoneType,
103         name: str | None = None,
104         retries: int = 1,
105         result_tool_name: str = 'final_result',
106         result_tool_description: str | None = None,
107         result_retries: int | None = None,
108         tools: Sequence[Tool[AgentDeps]] | ToolFuncEither[AgentDeps, ...] = (),
109         defer_model_check: bool = False,
110     ):
111         """Create an agent.
112
113         Args:
114             model: The default model to use for this agent, if not provide,
115                 you must provide the model when calling the agent.
116             result_type: The type of the result data, used to validate the result data, defaults to 'str'.
117             system_prompt: Static system prompts to use for this agent, you can also register system
118                 prompts via a function with ['system_prompt'][pydantic_ai.Agent.system_prompt].
119             deps_type: The type used for dependency injection, this parameter exists solely to allow you to fully
120                 parameterize the agent, and therefore get the best out of static type checking.
121                 If you're not using deps, but want type checking to pass, you can set 'deps=None' to satisfy Pyright
122                 or add a type hint ': Agent[None, <return type>]'.
123             name: The name of the agent, used for logging. If 'None', we try to infer the agent name from the call frame
124                 when the agent is first run.
125             retries: The default number of retries to allow before raising an error.
126             result_tool_name: The name of the tool to use for the final result.
127             result_tool_description: The description of the final result tool.
128             result_retries: The maximum number of retries to allow for result validation, defaults to 'retries'.
129             tools: Tools to register with the agent, you can also register tools via the decorators
130                 ['agent.tool'][pydantic_ai.Agent.tool] and ['agent.tool_plain'][pydantic_ai.Agent.tool_plain].
131             defer_model_check: by default, if you provide a [named][pydantic_ai.models.KnownModelName] model,
132                 it's evaluated to create a [Model][pydantic_ai.models.Model] instance immediately,
133                 which checks for the necessary environment variables. Set this to 'false'
134                 to defer the evaluation until the first run. Useful if you want to
135                 [override the model][pydantic_ai.Agent.override] for testing.
136
137         """
138         if model is None or defer_model_check:
139             self.model = model
140         else:
141             self.model = models.infer_model(model)
142
143         self.name = name
144         self._result_schema = _result.ResultSchema[result_type].build(
145             result_type, result_tool_name, result_tool_description
146         )
147         # if the result tool is None, or its schema allows 'str', we allow plain text results
148         self._allow_text_result = self._result_schema is None or self._result_schema.allow_text_result
149
150         self._system_prompts = (system_prompt,) if isinstance(system_prompt, str) else tuple(system_prompt)
151         self._function_tools = {}
152         self._default_retries = retries
153         for tool in tools:
154             if isinstance(tool, Tool):
155                 self._register_tool(tool)
156             else:
157                 self._register_tool(Tool(tool))
158         self._deps_type = deps_type
159         self._system_prompt_functions = []
160         self._max_result_retries = result_retries if result_retries is not None else retries
161         self._current_result_retry = 0
162         self._result_validators = []
163
164     async def run(
165         self,
166         user_prompt: str,
167         *,
168         message_history: list[_messages.Message] | None = None,
169         model: models.Model | models.KnownModelName | None = None,
170         deps: AgentDeps = None,
171         infer_name: bool = True,
172     ) -> result.RunResult[ResultData]:

```

```

171 """Run the agent with a user prompt in async mode.
172
173 Example:
174 ```py
175 from pydantic_ai import Agent
176
177 agent = Agent('openai:gpt-4o')
178
179 result_sync = agent.run_sync('What is the capital of Italy?')
180 print(result_sync.data)
181 #> Rome
182 ```
183
184 Args:
185     user_prompt: User input to start/continue the conversation.
186     message_history: History of the conversation so far.
187     model: Optional model to use for this run, required if 'model' was not set when creating the agent.
188     deps: Optional dependencies to use for this run.
189     infer_name: Whether to try to infer the agent name from the call frame if it's not set.
190
191 Returns:
192     The result of the run.
193 """
194 if infer_name and self.name is None:
195     self._infer_name(inspect.currentframe())
196 model_used, mode_selection = await self._get_model(model)
197
198 deps = self._get_deps(deps)
199
200 with _logfire.span(
201     '{agent_name} run {prompt=}',
202     prompt=user_prompt,
203     agent=self,
204     mode_selection=mode_selection,
205     model_name=model_used.name(),
206     agent_name=self.name or 'agent',
207 ) as run_span:
208     new_message_history, messages = await self._prepare_messages(deps, user_prompt, message_history)
209     self.last_run.messages = messages
210
211     for tool in self._function_tools.values():
212         tool.current_retry = 0
213
214     cost = result.Cost()
215
216     run_step = 0
217     while True:
218         run_step += 1
219         with _logfire.span('preparing model and tools {run_step=}', run_step=run_step):
220             agent_model = await self._prepare_model(model_used, deps)
221
222         with _logfire.span('model request', run_step=run_step) as model_req_span:
223             model_response, request_cost = await agent_model.request(messages)
224             model_req_span.set_attribute('response', model_response)
225             model_req_span.set_attribute('cost', request_cost)
226             model_req_span.message = f'model request -> {model_response.role}'
227
228         messages.append(model_response)
229         cost += request_cost
230
231         with _logfire.span('handle model response', run_step=run_step) as handle_span:
232             final_result, response_messages = await self._handle_model_response(model_response, deps)
233
234             # Add all messages to the conversation
235             messages.extend(response_messages)
236
237             # Check if we got a final result
238             if final_result is not None:
239                 result_data = final_result.data
240                 run_span.set_attribute('all_messages', messages)
241                 run_span.set_attribute('cost', cost)
242                 handle_span.set_attribute('result', result_data)
243                 handle_span.message = 'handle model response -> final result'
244                 return result.RunResult(messages, new_message_index, result_data, cost)
245             else:
246                 # continue the conversation
247                 handle_span.set_attribute('tool_responses', response_messages)
248                 response_msgs = ' '.join(r.role for r in response_messages)
249                 handle_span.message = f'handle model response -> {response_msgs}'
250
251 def run_sync(
252     self,
253     user_prompt: str,
254     *,
255     message_history: list[_messages.Message] | None = None,
256     model: models.Model | models.KnownModelName | None = None,
257     deps: AgentDeps = None,
258     infer_name: bool = True,
259 ) -> result.RunResult[ResultData]:
260     """Run the agent with a user prompt synchronously.
261
262     This is a convenience method that wraps `self.run` with `loop.run_until_complete()`.
263
264     Example:
265     ```py
266     from pydantic_ai import Agent
267
268     agent = Agent('openai:gpt-4o')
269
270     async def main():
271         result = await agent.run('What is the capital of France?')
272         print(result.data)
273         #> Paris
274     ```
275
276     Args:
277         user_prompt: User input to start/continue the conversation.
278         message_history: History of the conversation so far.
279         model: Optional model to use for this run, required if 'model' was not set when creating the agent.
280         deps: Optional dependencies to use for this run.
281         infer_name: Whether to try to infer the agent name from the call frame if it's not set.
282
283     Returns:
284         The result of the run.
285     """
286     if infer_name and self.name is None:
287         self._infer_name(inspect.currentframe())
288     loop = asyncio.get_event_loop()
289     return loop.run_until_complete(
290         self.run(user_prompt, message_history=message_history, model=model, deps=deps, infer_name=False)
291     )
292
293 @asynccontextmanager
294 async def run_stream(
295     self,
296     user_prompt: str,
297     *,
298     message_history: list[_messages.Message] | None = None,
299     model: models.Model | models.KnownModelName | None = None,
300     deps: AgentDeps = None,
301     infer_name: bool = True,

```

```

302 ) -> AsyncIterator[result.StreamedRunResult[AgentDeps, ResultData]]:
303     """Run the agent with a user prompt in async mode, returning a streamed response.
304
305     Example:
306     ```py
307     from pydantic_ai import Agent
308
309     agent = Agent('openai:gpt-4o')
310
311     async def main():
312         async with agent.run_stream('What is the capital of the UK?') as response:
313             print(await response.get_data())
314             #> London
315     ...
316
317     Args:
318         user_prompt: User input to start/continue the conversation.
319         message_history: History of the conversation so far.
320         model: Optional model to use for this run, required if 'model' was not set when creating the agent.
321         deps: Optional dependencies to use for this run.
322         infer_name: Whether to try to infer the agent name from the call frame if it's not set.
323
324     Returns:
325         The result of the run.
326     """
327     if infer_name and self.name is None:
328         # f_back because 'asynccontextmanager' adds one frame
329         if frame := inspect.currentframe(): # pragma: no branch
330             self._infer_name(frame.f_back)
331     model_used, mode_selection = await self._get_model(model)
332
333     deps = self._get_deps(deps)
334
335     with _logfire.span(
336         '{agent_name} run stream {prompt=}',
337         prompt=user_prompt,
338         agent=self,
339         mode_selection=mode_selection,
340         model_name=model_used.name(),
341         agent_name=self.name or 'agent',
342     ) as run_span:
343         new_message_index, messages = await self._prepare_messages(deps, user_prompt, message_history)
344         self.last_run_messages = messages
345
346         for tool in self._function_tools.values():
347             tool.current_retry = 0
348
349         cost = result.Cost()
350
351         run_step = 0
352         while True:
353             run_step += 1
354
355             with _logfire.span('preparing model and tools {run_step=}', run_step=run_step):
356                 agent_model = await self._prepare_model(model_used, deps)
357
358             with _logfire.span('model request {run_step=}', run_step=run_step) as model_req_span:
359                 async with agent_model.request_stream(messages) as model_response:
360                     model_req_span.set_attribute('response_type', model_response.__class__.__name__)
361                     # We want to end the "model request" span here, but we can't exit the context manager
362                     # in the traditional way
363                     model_req_span.__exit__(None, None, None)
364
365             with _logfire.span('handle model response') as handle_span:
366                 final_result, response_messages = await self._handle_streamed_model_response(
367                     model_response, deps
368                 )
369
370             # Add all messages to the conversation
371             messages.extend(response_messages)
372
373             # Check if we got a final result
374             if final_result is not None:
375                 result_stream = final_result.data
376                 run_span.set_attribute('all_messages', messages)
377                 handle_span.set_attribute('result_type', result_stream.__class__.__name__)
378                 handle_span.message = 'handle model response -> final result'
379                 yield result.StreamedRunResult(
380                     messages,
381                     new_message_index,
382                     cost,
383                     result_stream,
384                     self._result_schema,
385                     deps,
386                     self._result_validators,
387                     lambda m: run_span.set_attribute('all_messages', messages),
388                 )
389                 return
390             else:
391                 # continue the conversation
392                 handle_span.set_attribute('tool_responses', response_messages)
393                 response_msgs = ' '.join(r.role for r in response_messages)
394                 handle_span.message = f'handle model response -> {response_msgs}'
395                 # the model_response should have been fully streamed by now, we can add it's cost
396                 cost += model_response.cost()
397
398     @contextmanager
399     def override(
400         self,
401         *,
402         deps: AgentDeps | _utils.Unset = _utils.UNSET,
403         model: models.Model | models.KnownModelName | _utils.Unset = _utils.UNSET,
404     ) -> Iterator[None]:
405         """Context manager to temporarily override agent dependencies and model.
406
407         This is particularly useful when testing.
408         You can find an example of this [here](../testing-evals.md#overriding-model-via-pytest-fixtures).
409
410     Args:
411         deps: The dependencies to use instead of the dependencies passed to the agent run.
412         model: The model to use instead of the model passed to the agent run.
413     """
414     if _utils.is_set(deps):
415         override_deps_before = self._override_deps
416         self._override_deps = _utils.Some(deps)
417     else:
418         override_deps_before = _utils.UNSET
419
420     # noinspection PyTypeChecker
421     if _utils.is_set(model):
422         override_model_before = self._override_model
423         # noinspection PyTypeChecker
424         self._override_model = _utils.Some(models.infer_model(model)) # pyright: ignore[reportArgumentType]
425     else:
426         override_model_before = _utils.UNSET
427
428     try:
429         yield
430     finally:
431         if _utils.is_set(override_deps_before):
432             self._override_deps = override_deps_before

```

```

433     if _utils.is_set(override_model_before):
434         self._override_model = override_model_before
435
436 @overload
437 def system_prompt(
438     self, func: Callable[[RunContext[AgentDeps]], str], /
439 ) -> Callable[[RunContext[AgentDeps]], str]: ...
440
441 @overload
442 def system_prompt(
443     self, func: Callable[[RunContext[AgentDeps]], Awaitable[str]], /
444 ) -> Callable[[RunContext[AgentDeps]], Awaitable[str]]: ...
445
446 @overload
447 def system_prompt(self, func: Callable[[], str], /) -> Callable[[], str]: ...
448
449 @overload
450 def system_prompt(self, func: Callable[[], Awaitable[str]], /) -> Callable[[], Awaitable[str]]: ...
451
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
484
485 @overload
486 def result_validator(
487     self, func: Callable[[RunContext[AgentDeps], ResultData], ResultData], /
488 ) -> Callable[[RunContext[AgentDeps], ResultData], ResultData]: ...
489
490 @overload
491 def result_validator(
492     self, func: Callable[[RunContext[AgentDeps], ResultData], Awaitable[ResultData]], /
493 ) -> Callable[[RunContext[AgentDeps], ResultData], Awaitable[ResultData]]: ...
494
495 @overload
496 def result_validator(self, func: Callable[[ResultData], ResultData], /) -> Callable[[ResultData], ResultData]: ...
497
498 @overload
499 def result_validator(
500     self, func: Callable[[ResultData], Awaitable[ResultData]], /
501 ) -> Callable[[ResultData], Awaitable[ResultData]]: ...
502
503 def result_validator(
504     self, func: _result.ResultValidatorFunc[AgentDeps, ResultData], /
505 ) -> _result.ResultValidatorFunc[AgentDeps, ResultData]:
506     """Decorator to register a result validator function.
507
508     Optionally takes ['RunContext'] [pydantic_ai.tools.RunContext] as its first argument.
509     Can decorate a sync or async functions.
510
511     Overloads for every possible signature of 'result_validator' are included so the decorator doesn't obscure
512     the type of the function, see 'tests/typed_agent.py' for tests.
513
514     Example:
515     ```py
516     from pydantic_ai import Agent, ModelRetry, RunContext
517
518     agent = Agent('test', deps_type=str)
519
520     @agent.result_validator
521     def result_validator_simple(data: str) -> str:
522         if 'wrong' in data:
523             raise ModelRetry('wrong response')
524         return data
525
526     @agent.result_validator
527     async def result_validator_deps(ctx: RunContext[str], data: str) -> str:
528         if ctx.deps in data:
529             raise ModelRetry('wrong response')
530         return data
531
532     result = agent.run_sync('foobar', deps='spam')
533     print(result.data)
534     #> success (no tool calls)
535     ...
536     """
537     self._result_validators.append(_result.ResultValidator(func))
538     return func
539
540 @overload
541 def tool(self, func: ToolFuncContext[AgentDeps, ToolParams], /) -> ToolFuncContext[AgentDeps, ToolParams]: ...
542
543 @overload
544 def tool(
545     self,
546     /,
547     *,
548     retries: int | None = None,
549     prepare: ToolPrepareFunc[AgentDeps] | None = None,
550 ) -> Callable[[ToolFuncContext[AgentDeps, ToolParams]], ToolFuncContext[AgentDeps, ToolParams]]: ...
551
552 def tool(
553     self,
554     func: ToolFuncContext[AgentDeps, ToolParams] | None = None,
555     /,
556     *,
557     retries: int | None = None,
558     prepare: ToolPrepareFunc[AgentDeps] | None = None,
559 ) -> Any:
560     """Decorator to register a tool function which takes ['RunContext'] [pydantic_ai.tools.RunContext] as its first argument.
561
562     Can decorate a sync or async functions.
563

```

```

564 The docstring is inspected to extract both the tool description and description of each parameter,
565 [learn more](../agents.md#function-tools-and-schema).
566
567 We can't add overloads for every possible signature of tool, since the return type is a recursive union
568 so the signature of functions decorated with '@agent.tool' is obscured.
569
570 Example:
571 ```py
572 from pydantic_ai import Agent, RunContext
573
574 agent = Agent('test', deps_type=int)
575
576 @agent.tool
577 def foobar(ctx: RunContext[int], x: int) -> int:
578     return ctx.deps + x
579
580 @agent.tool(retries=2)
581 async def spam(ctx: RunContext[str], y: float) -> float:
582     return ctx.deps + y
583
584 result = agent.run_sync('foobar', deps=1)
585 print(result.data)
586 #> {"foobar":1,"spam":1.0}
587 ...
588
589 Args:
590     func: The tool function to register.
591     retries: The number of retries to allow for this tool, defaults to the agent's default retries,
592             which defaults to 1.
593     prepare: custom method to prepare the tool definition for each step, return 'None' to omit this
594             tool from a given step. This is useful if you want to customise a tool at call time,
595             or omit it completely from a step. See ['ToolPrepareFunc'](pydantic_ai.tools.ToolPrepareFunc).
596 """
597 if func is None:
598
599     def tool_decorator(
600         func_: ToolFuncContext[AgentDeps, ToolParams],
601     ) -> ToolFuncContext[AgentDeps, ToolParams]:
602         # noinspection PyTypeChecker
603         self._register_function(func_, True, retries, prepare)
604         return func_
605
606     return tool_decorator
607 else:
608     # noinspection PyTypeChecker
609     self._register_function(func, True, retries, prepare)
610     return func
611
612 @overload
613 def tool_plain(self, func: ToolFuncPlain[ToolParams], /) -> ToolFuncPlain[ToolParams]: ...
614
615 @overload
616 def tool_plain(
617     self,
618     /,
619     *,
620     retries: int | None = None,
621     prepare: ToolPrepareFunc[AgentDeps] | None = None,
622 ) -> Callable[[ToolFuncPlain[ToolParams]], ToolFuncPlain[ToolParams]]: ...
623
624 def tool_plain(
625     self,
626     func: ToolFuncPlain[ToolParams] | None = None,
627     /,
628     *,
629     retries: int | None = None,
630     prepare: ToolPrepareFunc[AgentDeps] | None = None,
631 ) -> Any:
632     """Decorator to register a tool function which DOES NOT take 'RunContext' as an argument.
633
634     Can decorate a sync or async functions.
635
636     The docstring is inspected to extract both the tool description and description of each parameter,
637     [learn more](../agents.md#function-tools-and-schema).
638
639     We can't add overloads for every possible signature of tool, since the return type is a recursive union
640     so the signature of functions decorated with '@agent.tool' is obscured.
641
642     Example:
643     ```py
644     from pydantic_ai import Agent, RunContext
645
646     agent = Agent('test')
647
648     @agent.tool
649     def foobar(ctx: RunContext[int]) -> int:
650         return 123
651
652     @agent.tool(retries=2)
653     async def spam(ctx: RunContext[str]) -> float:
654         return 3.14
655
656     result = agent.run_sync('foobar', deps=1)
657     print(result.data)
658     #> {"foobar":123,"spam":3.14}
659     ...
660
661     Args:
662         func: The tool function to register.
663         retries: The number of retries to allow for this tool, defaults to the agent's default retries,
664                 which defaults to 1.
665         prepare: custom method to prepare the tool definition for each step, return 'None' to omit this
666                 tool from a given step. This is useful if you want to customise a tool at call time,
667                 or omit it completely from a step. See ['ToolPrepareFunc'](pydantic_ai.tools.ToolPrepareFunc).
668     """
669     if func is None:
670
671         def tool_decorator(func_: ToolFuncPlain[ToolParams]) -> ToolFuncPlain[ToolParams]:
672             # noinspection PyTypeChecker
673             self._register_function(func_, False, retries, prepare)
674             return func_
675
676         return tool_decorator
677     else:
678         self._register_function(func, False, retries, prepare)
679         return func
680
681 def _register_function(
682     self,
683     func: ToolFuncEither[AgentDeps, ToolParams],
684     takes_ctx: bool,
685     retries: int | None,
686     prepare: ToolPrepareFunc[AgentDeps] | None,
687 ) -> None:
688     """Private utility to register a function as a tool."""
689     retries_ = retries if retries is not None else self._default_retries
690     tool = Tool(func, takes_ctx=takes_ctx, max_retries=retries_, prepare=prepare)
691     self._register_tool(tool)
692
693 def _register_tool(self, tool: Tool[AgentDeps]) -> None:
694     """Private utility to register a tool instance."""

```

```

695 if tool.max_retries is None:
696     # noinspection PyTypeChecker
697     tool = dataclasses.replace(tool, max_retries=self._default_retries)
698
699 if tool.name in self._function_tools:
700     raise exceptions.UserError(f'Tool name conflicts with existing tool: {tool.name!r}')
701
702 if self._result_schema and tool.name in self._result_schema.tools:
703     raise exceptions.UserError(f'Tool name conflicts with result schema name: {tool.name!r}')
704
705 self._function_tools[tool.name] = tool
706
707 async def _get_model(self, model: models.Model | models.KnownModelName | None) -> tuple[models.Model, str]:
708     """Create a model configured for this agent.
709
710     Args:
711         model: model to use for this run, required if 'model' was not set when creating the agent.
712
713     Returns:
714         a tuple of '(model used, how the model was selected)'
715     """
716     model_: models.Model
717     if some_model := self._override_model:
718         # we don't want 'override()' to cover up errors from the model not being defined, hence this check
719         if model is None and self.model is None:
720             raise exceptions.UserError(
721                 'model' must be set either when creating the agent or when calling it. '
722                 '(Even when 'override(model=...)' is customizing the model that will actually be called)'
723             )
724         model_ = some_model.value
725         mode_selection = 'override-model'
726     elif model is not None:
727         model_ = models.infer_model(model)
728         mode_selection = 'custom'
729     elif self.model is not None:
730         # noinspection PyTypeChecker
731         model_ = self.model = models.infer_model(self.model)
732         mode_selection = 'from-agent'
733     else:
734         raise exceptions.UserError('model' must be set either when creating the agent or when calling it.')
735
736     return model_, mode_selection
737
738 async def _prepare_model(self, model: models.Model, deps: AgentDeps) -> models.AgentModel:
739     """Create building tools and create an agent model."""
740     function_tools: list[ToolDefinition] = []
741
742     async def add_tool(tool: Tool[AgentDeps]) -> None:
743         ctx = RunContext(deps, tool.current_retry, tool.name)
744         if tool_def := await tool.prepare_tool_def(ctx):
745             function_tools.append(tool_def)
746
747     await asyncio.gather(*map(add_tool, self._function_tools.values()))
748
749     return await model.agent_model(
750         function_tools=function_tools,
751         allow_text_result=self._allow_text_result,
752         result_tools=self._result_schema.tool_defs() if self._result_schema is not None else [],
753     )
754
755 async def _prepare_messages(
756     self, deps: AgentDeps, user_prompt: str, message_history: list[_messages.Message] | None
757 ) -> tuple[int, list[_messages.Message]]:
758     # if message history includes system prompts, we don't want to regenerate them
759     if message_history and any(m.role == 'system' for m in message_history):
760         # shallow copy messages
761         messages = message_history.copy()
762     else:
763         messages = await self._init_messages(deps)
764         if message_history:
765             messages += message_history
766
767     new_message_index = len(messages)
768     messages.append(_messages.UserPrompt(user_prompt))
769     return new_message_index, messages
770
771 async def _handle_model_response(
772     self, model_response: _messages.ModelAnyResponse, deps: AgentDeps
773 ) -> tuple[_MarkFinalResult[ResultData] | None, list[_messages.Message]]:
774     """Process a non-streamed response from the model.
775
776     Returns:
777         A tuple of '(final_result, messages)'. If 'final_result' is not 'None', the conversation should end.
778     """
779     if model_response.role == 'model-text-response':
780         # plain string response
781         if self._allow_text_result:
782             result_data_input = cast(ResultData, model_response.content)
783             try:
784                 result_data = await self._validate_result(result_data_input, deps, None)
785             except _result.ToolRetryError as e:
786                 self._incr_result_retry()
787                 return None, [e.tool_retry]
788             else:
789                 return _MarkFinalResult(result_data), []
790         else:
791             self._incr_result_retry()
792             response = _messages.RetryPrompt(
793                 content='Plain text responses are not permitted, please call one of the functions instead.',
794             )
795             return None, [response]
796     elif model_response.role == 'model-structured-response':
797         if self._result_schema is not None:
798             # if there's a result schema, and any of the calls match one of its tools, return the result
799             # NOTE: this means we ignore any other tools called here
800             if match := self._result_schema.find_tool(model_response):
801                 call, result_tool = match
802                 try:
803                     result_data = result_tool.validate(call)
804                     result_data = await self._validate_result(result_data, deps, call)
805                 except _result.ToolRetryError as e:
806                     self._incr_result_retry()
807                     return None, [e.tool_retry]
808                 else:
809                     # Add a ToolReturn message for the schema tool call
810                     tool_return = _messages.ToolReturn(
811                         tool_name=call.tool_name,
812                         content='Final result processed.',
813                         tool_call_id=call.tool_call_id,
814                     )
815                     return _MarkFinalResult(result_data), [tool_return]
816
817     if not model_response.calls:
818         raise exceptions.UnexpectedModelBehavior('Received empty tool call message')
819
820     # otherwise we run all tool functions in parallel
821     messages: list[_messages.Message] = []
822     tasks: list[asyncio.Task[_messages.Message]] = []
823     for call in model_response.calls:
824         if tool := self._function_tools.get(call.tool_name):
825             tasks.append(asyncio.create_task(tool.run(deps, call), name=call.tool_name))

```

```

826         else:
827             messages.append(self._unknown_tool(call.tool_name))
828
829         with _logfire.span('running {tools=}', tools=[t.get_name() for t in tasks]):
830             task_results: Sequence[_messages.Message] = await asyncio.gather(*tasks)
831             messages.extend(task_results)
832             return None, messages
833         else:
834             assert_never(model_response)
835
836     async def _handle_streamed_model_response(
837         self, model_response: models.EitherStreamedResponse, deps: AgentDeps
838     ) -> tuple[_MarkFinalResult[models.EitherStreamedResponse] | None, list[_messages.Message]]:
839         """Process a streamed response from the model.
840
841         Returns:
842             A tuple of (final_result, messages). If final_result is not None, the conversation should end.
843         """
844         if isinstance(model_response, models.StreamTextResponse):
845             # plain string response
846             if self._allow_text_result:
847                 return _MarkFinalResult(model_response), []
848             else:
849                 self._incr_result_retry()
850                 response = _messages.RetryPrompt(
851                     content='Plain text responses are not permitted, please call one of the functions instead.',
852                 )
853                 # stream the response, so cost is correct
854                 async for _ in model_response:
855                     pass
856
857                 return None, [response]
858         else:
859             assert isinstance(model_response, models.StreamStructuredResponse), f'Unexpected response: {model_response}'
860             if self._result_schema is not None:
861                 # if there's a result schema, iterate over the stream until we find at least one tool
862                 # NOTE: this means we ignore any other tools called here
863                 structured_msg = model_response.get()
864                 while not structured_msg.calls:
865                     try:
866                         await model_response._anext__()
867                     except StopAsyncIteration:
868                         break
869                     structured_msg = model_response.get()
870
871                 if match := self._result_schema.find_tool(structured_msg):
872                     call, _ = match
873                     tool_return = _messages.ToolReturn(
874                         tool_name=call.tool_name,
875                         content='Final result processed.',
876                         tool_call_id=call.tool_call_id,
877                     )
878                     return _MarkFinalResult(model_response), [tool_return]
879
880             # the model is calling a tool function, consume the response to get the next message
881             async for _ in model_response:
882                 pass
883             structured_msg = model_response.get()
884             if not structured_msg.calls:
885                 raise exceptions.UnexpectedModelBehavior('Received empty tool call message')
886             messages: list[_messages.Message] = [structured_msg]
887
888             # we now run all tool functions in parallel
889             tasks: list[asyncio.Task[_messages.Message]] = []
890             for call in structured_msg.calls:
891                 if tool := self._function_tools.get(call.tool_name):
892                     tasks.append(asyncio.create_task(tool.run(deps, call), name=call.tool_name))
893                 else:
894                     messages.append(self._unknown_tool(call.tool_name))
895
896             with _logfire.span('running {tools=}', tools=[t.get_name() for t in tasks]):
897                 task_results: Sequence[_messages.Message] = await asyncio.gather(*tasks)
898                 messages.extend(task_results)
899                 return None, messages
900
901     async def _validate_result(
902         self, result_data: ResultData, deps: AgentDeps, tool_call: _messages.ToolCall | None
903     ) -> ResultData:
904         for validator in self._result_validators:
905             result_data = await validator.validate(result_data, deps, self._current_result_retry, tool_call)
906         return result_data
907
908     def _incr_result_retry(self) -> None:
909         self._current_result_retry += 1
910         if self._current_result_retry > self._max_result_retries:
911             raise exceptions.UnexpectedModelBehavior(
912                 f'Exceeded maximum retries ({self._max_result_retries}) for result validation'
913             )
914
915     async def _init_messages(self, deps: AgentDeps) -> list[_messages.Message]:
916         """Build the initial messages for the conversation."""
917         messages: list[_messages.Message] = [_messages.SystemPrompt(p) for p in self._system_prompts]
918         for sys_prompt_runner in self._system_prompt_functions:
919             prompt = await sys_prompt_runner.run(deps)
920             messages.append(_messages.SystemPrompt(prompt))
921         return messages
922
923     def _unknown_tool(self, tool_name: str) -> _messages.RetryPrompt:
924         self._incr_result_retry()
925         names = list(self._function_tools.keys())
926         if self._result_schema:
927             names.extend(self._result_schema.tool_names())
928         if names:
929             msg = f'Available tools: {', '.join(names)}'
930         else:
931             msg = 'No tools available.'
932         return _messages.RetryPrompt(content=f'Unknown tool name: {tool_name!r}. {msg}')
933
934     def _get_deps(self, deps: AgentDeps) -> AgentDeps:
935         """Get deps for a run.
936
937         If we've overridden deps via '_override_deps', use that, otherwise use the deps passed to the call.
938
939         We could do runtime type checking of deps against 'self._deps_type', but that's a slippery slope.
940         """
941         if some_deps := self._override_deps:
942             return some_deps.value
943         else:
944             return deps
945
946     def _infer_name(self, function_frame: FrameType | None) -> None:
947         """Infer the agent name from the call frame.
948
949         Usage should be 'self._infer_name(inspect.currentframe())'.
950         """
951         assert self.name is None, 'Name already set'
952         if function_frame is not None: # pragma: no branch
953             if parent_frame := function_frame.f_back: # pragma: no branch
954                 for name, item in parent_frame.f_locals.items():
955                     if item is self:
956                         self.name = name

```



```
957         return
958     if parent_frame.f_locals != parent_frame.f_globals:
959         # if we couldn't find the agent in locals and globals are a different dict, try globals
960         for name, item in parent_frame.f_globals.items():
961             if item is self:
962                 self.name = name
963                 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
<code>model</code>	<code>Model   KnownModelName   None</code>	The default model to use for this agent, if not provide, you must provide the model when calling the agent.	<code>None</code>
<code>result_type</code>	<code>type[ResultData]</code>	The type of the result data, used to validate the result data, defaults to <code>str</code> .	<code>str</code>
<code>system_prompt</code>	<code>str   Sequence[str]</code>	Static system prompts to use for this agent, you can also register system prompts via a function with <code>system_prompt</code> .	<code>()</code>
<code>deps_type</code>	<code>type[AgentDeps]</code>	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 <code>deps=None</code> to satisfy Pyright or add a type hint : <code>Agent[None, &lt;return type&gt;]</code> .	<code>NoneType</code>
<code>name</code>	<code>str   None</code>	The name of the agent, used for logging. If <code>None</code> , we try to infer the agent name from the call frame when the agent is first run.	<code>None</code>
<code>retries</code>	<code>int</code>	The default number of retries to allow before raising an error.	<code>1</code>
<code>result_tool_name</code>	<code>str</code>	The name of the tool to use for the final result.	<code>'final_result'</code>
<code>result_tool_description</code>	<code>str   None</code>	The description of the final result tool.	<code>None</code>
<code>result_retries</code>	<code>int   None</code>	The maximum number of retries to allow for result validation, defaults to <code>retries</code> .	<code>None</code>
<code>tools</code>	<code>Sequence[Tool[AgentDeps]   ToolFuncEither[AgentDeps, ...]]</code>	Tools to register with the agent, you can also register tools via the decorators <code>@agent.tool</code> and <code>@agent.tool_plain</code> .	<code>()</code>
<code>defer_model_check</code>	<code>bool</code>	by default, if you provide a <b>named</b> model, it's evaluated to create a <code>Model</code> instance immediately, which checks for the necessary environment variables. Set this to <code>false</code> to defer the evaluation until the first run. Useful if you want to <b>override the model</b> for testing.	<code>False</code>

```
95 def __init__(
96     self,
97     model: models.Model | models.KnownModelName | None = None,
98     *,
99     result_type: type[ResultData] = str,
100     system_prompt: str | Sequence[str] = (),
101     deps_type: type[AgentDeps] = NoneType,
102     name: str | None = None,
103     retries: int = 1,
104     result_tool_name: str = 'final_result',
105     result_tool_description: str | None = None,
106     result_retries: int | None = None,
107     tools: Sequence[Tool[AgentDeps] | ToolFuncEither[AgentDeps, ...]] = (),
108     defer_model_check: bool = False,
109 ):
110     """Create an agent.
111
112     Args:
113         model: The default model to use for this agent, if not provide,
114             you must provide the model when calling the agent.
115         result_type: The type of the result data, used to validate the result data, defaults to 'str'.
116         system_prompt: Static system prompts to use for this agent, you can also register system
117             prompts via a function with ['system_prompt'][pydantic_ai.Agent.system_prompt].
118         deps_type: The type used for dependency injection, this parameter exists solely to allow you to fully
119             parameterize the agent, and therefore get the best out of static type checking.
120             If you're not using deps, but want type checking to pass, you can set 'deps=None' to satisfy Pyright
121             or add a type hint ': Agent[None, <return type>]'.
122         name: The name of the agent, used for logging. If 'None', we try to infer the agent name from the call frame
123             when the agent is first run.
124         retries: The default number of retries to allow before raising an error.
125         result_tool_name: The name of the tool to use for the final result.
126         result_tool_description: The description of the final result tool.
127         result_retries: The maximum number of retries to allow for result validation, defaults to 'retries'.
128         tools: Tools to register with the agent, you can also register tools via the decorators
129             ['@agent.tool'][pydantic_ai.Agent.tool] and ['@agent.tool_plain'][pydantic_ai.Agent.tool_plain].
130         defer_model_check: by default, if you provide a [named][pydantic_ai.models.KnownModelName] model,
131             it's evaluated to create a ['Model'][pydantic_ai.models.Model] instance immediately,
132             which checks for the necessary environment variables. Set this to 'false'
133             to defer the evaluation until the first run. Useful if you want to
134             [override the model][pydantic_ai.Agent.override] for testing.
135
136     """
137     if model is None or defer_model_check:
138         self.model = model
139     else:
140         self.model = models.infer_model(model)
141
142     self.name = name
143     self._result_schema = _result.ResultSchema[result_type].build(
144         result_type, result_tool_name, result_tool_description
145     )
146     # if the result tool is None, or its schema allows 'str', we allow plain text results
147     self._allow_text_result = self._result_schema is None or self._result_schema.allow_text_result
148
149     self._system_prompts = (system_prompt,) if isinstance(system_prompt, str) else tuple(system_prompt)
150     self._function_tools = {}
151     self._default_retries = retries
152     for tool in tools:
153         if isinstance(tool, Tool):
154             self._register_tool(tool)
155         else:
156             self._register_tool(Tool(tool))
157     self._deps_type = deps_type
158     self._system_prompt_functions = []
159     self._max_result_retries = result_retries if result_retries is not None else retries
160     self._current_result_retry = 0
161     self._result_validators = []
```

name instance-attribute

```
name: str | None = name
```

The name of the agent, used for logging.

If `None`, we try to infer the agent name from the call frame when the agent is first run.

run async

```
run(
    user_prompt: str,
    *,
    message_history: list[Message] | None = None,
    model: Model | KnownModelName | None = None,
    deps: AgentDeps = None,
    infer_name: bool = True
) -> RunResult[ResultData]
```

Run the agent with a user prompt in async mode.

Example:

```
from pydantic_ai import Agent

agent = Agent('openai:gpt-4o')

result_sync = agent.run_sync('What is the capital of Italy?')
print(result_sync.data)
#> Rome
```

Parameters:

Name	Type	Description	Default
user_prompt	str	User input to start/continue the conversation.	required
message_history	list[Message]   None	History of the conversation so far.	None
model	Model   KnownModelName   None	Optional model to use for this run, required if <code>model</code> was not set when creating the agent.	None
deps	AgentDeps	Optional dependencies to use for this run.	None

Name	Type	Description	Default
<code>infer_name</code>	<code>bool</code>	Whether to try to infer the agent name from the call frame if it's not set.	<code>True</code>

Returns:

Type	Description
<code>RunResult[ResultData]</code>	The result of the run.

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

```
162 async def run(
163     self,
164     user_prompt: str,
165     *,
166     message_history: list[_messages.Message] | None = None,
167     model: models.Model | models.KnownModelName | None = None,
168     deps: AgentDeps = None,
169     infer_name: bool = True,
170 ) -> result.RunResult[ResultData]:
171     """Run the agent with a user prompt in async mode.
172
173     Example:
174     ```py
175     from pydantic_ai import Agent
176
177     agent = Agent('openai:gpt-4o')
178
179     result_sync = agent.run_sync('What is the capital of Italy?')
180     print(result_sync.data)
181     #> Rome
182     ```
183
184     Args:
185         user_prompt: User input to start/continue the conversation.
186         message_history: History of the conversation so far.
187         model: Optional model to use for this run, required if 'model' was not set when creating the agent.
188         deps: Optional dependencies to use for this run.
189         infer_name: Whether to try to infer the agent name from the call frame if it's not set.
190
191     Returns:
192         The result of the run.
193     """
194     if infer_name and self.name is None:
195         self._infer_name(inspect.currentframe())
196     model_used, mode_selection = await self._get_model(model)
197
198     deps = self._get_deps(deps)
199
200     with _logfire.span(
201         '{agent_name} run {prompt=}',
202         prompt=user_prompt,
203         agent=self,
204         mode_selection=mode_selection,
205         model_name=model_used.name(),
206         agent_name=self.name or 'agent',
207     ) as run_span:
208         new_message_index, messages = await self._prepare_messages(deps, user_prompt, message_history)
209         self.last_run_messages = messages
210
211         for tool in self._function_tools.values():
212             tool.current_retry = 0
213
214         cost = result.Cost()
215
216         run_step = 0
217         while True:
218             run_step += 1
219             with _logfire.span('preparing model and tools {run_step=}', run_step=run_step):
220                 agent_model = await self._prepare_model(model_used, deps)
221
222             with _logfire.span('model request', run_step=run_step) as model_req_span:
223                 model_response, request_cost = await agent_model.request(messages)
224                 model_req_span.set_attribute('response', model_response)
225                 model_req_span.set_attribute('cost', request_cost)
226                 model_req_span.message = f'model request -> {model_response.role}'
227
228             messages.append(model_response)
229             cost += request_cost
230
231             with _logfire.span('handle model response', run_step=run_step) as handle_span:
232                 final_result, response_messages = await self._handle_model_response(model_response, deps)
233
234                 # Add all messages to the conversation
235                 messages.extend(response_messages)
236
237                 # Check if we got a final result
238                 if final_result is not None:
239                     result_data = final_result.data
240                     run_span.set_attribute('all_messages', messages)
241                     run_span.set_attribute('cost', cost)
242                     handle_span.set_attribute('result', result_data)
243                     handle_span.message = 'handle model response -> final result'
244                     return result.RunResult(messages, new_message_index, result_data, cost)
245                 else:
246                     # continue the conversation
247                     handle_span.set_attribute('tool_responses', response_messages)
248                     response_msgs = ' '.join(r.role for r in response_messages)
249                     handle_span.message = f'handle model response -> {response_msgs}'
```

## run\_sync

```
run_sync(
    user_prompt: str,
    *,
    message_history: list[Message] | None = None,
    model: Model | KnownModelName | None = None,
    deps: AgentDeps = None,
    infer_name: bool = True
) -> RunResult[ResultData]
```

Run the agent with a user prompt synchronously.

This is a convenience method that wraps `self.run` with `loop.run_until_complete()`.

Example:

```
from pydantic_ai import Agent

agent = Agent('openai:gpt-4o')

async def main():
    result = await agent.run('What is the capital of France?')
    print(result.data)
    #> Paris
```

Parameters:

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

Returns:

Type	Description
<code>RunResult[ResultData]</code>	The result of the run.

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

```
251 def run_sync(
252     self,
253     user_prompt: str,
254     *,
255     message_history: list[Message] | None = None,
256     model: Model | KnownModelName | None = None,
257     deps: AgentDeps = None,
258     infer_name: bool = True,
259 ) -> RunResult[ResultData]:
260     """Run the agent with a user prompt synchronously.
261
262     This is a convenience method that wraps 'self.run' with 'loop.run_until_complete()'.
263
264     Example:
265     ```py
266     from pydantic_ai import Agent
267
268     agent = Agent('openai:gpt-4o')
269
270     async def main():
271         result = await agent.run('What is the capital of France?')
272         print(result.data)
273         #> Paris
274     ...
275
276     Args:
277         user_prompt: User input to start/continue the conversation.
278         message_history: History of the conversation so far.
279         model: Optional model to use for this run, required if 'model' was not set when creating the agent.
280         deps: Optional dependencies to use for this run.
281         infer_name: Whether to try to infer the agent name from the call frame if it's not set.
282
283     Returns:
284         The result of the run.
285     """
286     if infer_name and self.name is None:
287         self._infer_name(inspect.currentframe())
288     loop = asyncio.get_event_loop()
289     return loop.run_until_complete(
290         self.run(user_prompt, message_history=message_history, model=model, deps=deps, infer_name=False)
291     )
```

`run_stream` *async*

```
run_stream(
    user_prompt: str,
    *,
    message_history: list[Message] | None = None,
    model: Model | KnownModelName | None = None,
    deps: AgentDeps = None,
    infer_name: bool = True
) -> AsyncIterator[
    StreamedRunResult[AgentDeps, ResultData]
]
```

Run the agent with a user prompt in async mode, returning a streamed response.

Example:

```
from pydantic_ai import Agent

agent = Agent('openai:gpt-4o')

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

Parameters:

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

Returns:

Type	Description
<code>AsyncIterator[StreamedRunResult[AgentDeps, ResultData]]</code>	The result of the run.

```

293 @asynccontextmanager
294 async def run_stream(
295     self,
296     user_prompt: str,
297     *,
298     message_history: list[_messages.Message] | None = None,
299     model: models.Model | models.KnownModelName | None = None,
300     deps: AgentDeps = None,
301     infer_name: bool = True,
302 ) -> AsyncIterator[result.StreamedRunResult[AgentDeps, ResultData]]:
303     """Run the agent with a user prompt in async mode, returning a streamed response.
304
305     Example:
306     ```py
307     from pydantic_ai import Agent
308
309     agent = Agent('openai:gpt-4o')
310
311     async def main():
312         async with agent.run_stream('What is the capital of the UK?') as response:
313             print(await response.get_data())
314         #> London
315     ```
316
317     Args:
318         user_prompt: User input to start/continue the conversation.
319         message_history: History of the conversation so far.
320         model: Optional model to use for this run, required if 'model' was not set when creating the agent.
321         deps: Optional dependencies to use for this run.
322         infer_name: Whether to try to infer the agent name from the call frame if it's not set.
323
324     Returns:
325         The result of the run.
326     """
327     if infer_name and self.name is None:
328         # f_back because 'asynccontextmanager' adds one frame
329         if frame := inspect.currentframe(): # pragma: no branch
330             self._infer_name(frame.f_back)
331     model_used, mode_selection = await self._get_model(model)
332
333     deps = self._get_deps(deps)
334
335     with _logfire.span(
336         '{agent_name} run_stream {prompt=}',
337         prompt=user_prompt,
338         agent=self,
339         mode_selection=mode_selection,
340         model_name=model_used.name(),
341         agent_name=self.name or 'agent',
342     ) as run_span:
343         new_message_index, messages = await self._prepare_messages(deps, user_prompt, message_history)
344         self.last_run_messages = messages
345
346         for tool in self._function_tools.values():
347             tool.current_retry = 0
348
349         cost = result.Cost()
350
351         run_step = 0
352         while True:
353             run_step += 1
354
355             with _logfire.span('preparing model and tools {run_step=}', run_step=run_step):
356                 agent_model = await self._prepare_model(model_used, deps)
357
358             with _logfire.span('model request {run_step=}', run_step=run_step) as model_req_span:
359                 async with agent_model.request_stream(messages) as model_response:
360                     model_req_span.set_attribute('response_type', model_response.__class__.__name__)
361                     # We want to end the "model request" span here, but we can't exit the context manager
362                     # in the traditional way
363                     model_req_span.__exit__(None, None, None)
364
365             with _logfire.span('handle model response') as handle_span:
366                 final_result, response_messages = await self._handle_streamed_model_response(
367                     model_response, deps
368                 )
369
370             # Add all messages to the conversation
371             messages.extend(response_messages)
372
373             # Check if we got a final result
374             if final_result is not None:
375                 result_stream = final_result.data
376                 run_span.set_attribute('all_messages', messages)
377                 handle_span.set_attribute('result_type', result_stream.__class__.__name__)
378                 handle_span.message = 'handle model response -> final result'
379                 yield result.StreamedRunResult(
380                     messages,
381                     new_message_index,
382                     cost,
383                     result_stream,
384                     self._result_schema,
385                     deps,
386                     self._result_validators,
387                     lambda m: run_span.set_attribute('all_messages', messages),
388                 )
389                 return
390             else:
391                 # continue the conversation
392                 handle_span.set_attribute('tool_responses', response_messages)
393                 response_msgs = ' '.join(r.role for r in response_messages)
394                 handle_span.message = f'handle model response -> {response_msgs}'
395                 # the model_response should have been fully streamed by now, we can add it's cost
396                 cost += model_response.cost()

```

model instance-attribute

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

The default model configured for this agent.

override

```

override(
    *,
    deps: AgentDeps | Unset = UNSET,
    model: Model | KnownModelName | Unset = UNSET
) -> Iterator[None]

```

Context manager to temporarily override agent dependencies and model.

This is particularly useful when testing. You can find an example of this [here](#).

Parameters:

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

```
99 Source code in pydantic_ai_slim/pydantic_ai/agent.py
398 @contextmanager
399 def override(
400     self,
401     *,
402     deps: AgentDeps | _utils.Unset = _utils.UNSET,
403     model: models.Model | models.KnownModelName | _utils.Unset = _utils.UNSET,
404 ) -> Iterator[None]:
405     """Context manager to temporarily override agent dependencies and model.
406
407     This is particularly useful when testing.
408     You can find an example of this [here](../testing-evals.md#overriding-model-via-pytest-fixtures).
409
410     Args:
411         deps: The dependencies to use instead of the dependencies passed to the agent run.
412         model: The model to use instead of the model passed to the agent run.
413     """
414     if _utils.is_set(deps):
415         override_deps_before = self._override_deps
416         self._override_deps = _utils.Some(deps)
417     else:
418         override_deps_before = _utils.UNSET
419
420     # noinspection PyTypeChecker
421     if _utils.is_set(model):
422         override_model_before = self._override_model
423         # noinspection PyTypeChecker
424         self._override_model = _utils.Some(models.infer_model(model)) # pyright: ignore[reportArgumentType]
425     else:
426         override_model_before = _utils.UNSET
427
428     try:
429         yield
430     finally:
431         if _utils.is_set(override_deps_before):
432             self._override_deps = override_deps_before
433         if _utils.is_set(override_model_before):
434             self._override_model = override_model_before
```

`last_run_messages` class-attribute instance-attribute

```
last_run_messages: list[Message] | None = None
```

The messages from the last run, useful when a run raised an exception.

Note: these are not used by the agent, e.g. in future runs, they are just stored for developers' convenience.

system\_prompt

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

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

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

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

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

Decorator to register a system prompt function.

Optionally takes `RunContext` as its only argument. Can decorate a sync or async functions.

Overloads for every possible signature of `system_prompt` are included so the decorator doesn't obscure the type of the function, see `tests/typed_agent.py` for tests.

Example:

```
from pydantic_ai import Agent, RunContext

agent = Agent('test', deps_type=str)

@agent.system_prompt
def simple_system_prompt() -> str:
    return 'foobar'

@agent.system_prompt
async def async_system_prompt(ctx: RunContext[str]) -> str:
    return f'{ctx.deps} is the best'

result = agent.run_sync('foobar', deps='spam')
print(result.data)
#> success (no tool calls)
```

```
99 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     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	<code>ToolFuncContext[AgentDeps, ToolParams]   None</code>	The tool function to register.	None
retries	<code>int   None</code>	The number of retries to allow for this tool, defaults to the agent's default retries, which defaults to 1.	None
prepare	<code>ToolPrepareFunc[AgentDeps]   None</code>	custom method to prepare the tool definition for each step, return <code>None</code> 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 <code>ToolPrepareFunc</code> .	None



```

552 def tool(
553     self,
554     func: ToolFuncContext[AgentDeps, ToolParams] | None = None,
555     /,
556     *,
557     retries: int | None = None,
558     prepare: ToolPrepareFunc[AgentDeps] | None = None,
559 ) -> Any:
560     """Decorator to register a tool function which takes ['RunContext']  

561     [pydantic_ai.tools.RunContext] as its first argument.
562
563     Can decorate a sync or async functions.
564
565     The docstring is inspected to extract both the tool description and description of each parameter,  

566     [learn more](../agents.md#function-tools-and-schema).
567
568     We can't add overloads for every possible signature of tool, since the return type is a recursive union  

569     so the signature of functions decorated with '@agent.tool' is obscured.
570
571     Example:
572     ```py
573     from pydantic_ai import Agent, RunContext
574
575     agent = Agent('test', deps_type=int)
576
577     @agent.tool
578     def foobar(ctx: RunContext[int], x: int) -> int:
579         return ctx.deps + x
580
581     @agent.tool(retries=2)
582     async def spam(ctx: RunContext[str], y: float) -> float:
583         return ctx.deps + y
584
585     result = agent.run_sync('foobar', deps=1)
586     print(result.data)
587     #> {"foobar":1,"spam":1.0}
588     ...
589
590     Args:
591     func: The tool function to register.
592     retries: The number of retries to allow for this tool, defaults to the agent's default retries,  

593     which defaults to 1.
594     prepare: custom method to prepare the tool definition for each step, return 'None' to omit this  

595     tool from a given step. This is useful if you want to customise a tool at call time,  

596     or omit it completely from a step. See ['ToolPrepareFunc']  

597     [pydantic_ai.tools.ToolPrepareFunc].
598     """
599     if func is None:
600         def tool_decorator(
601             func_: ToolFuncContext[AgentDeps, ToolParams],
602         ) -> ToolFuncContext[AgentDeps, ToolParams]:
603             # noinspection PyTypeChecker
604             self._register_function(func_, True, retries, prepare)
605             return func_
606         return tool_decorator
607     else:
608         # noinspection PyTypeChecker
609         self._register_function(func, True, retries, prepare)
610         return func

```

## tool\_plain

```

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

```

```

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

```

```

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

```

Decorator to register a tool function which DOES NOT take `RunContext` as an argument.

Can decorate a sync or async functions.

The docstring is inspected to extract both the tool description and description of each parameter, [learn more](#).

We can't add overloads for every possible signature of tool, since the return type is a recursive union so the signature of functions decorated with `@agent.tool` is obscured.

Example:

```

from pydantic_ai import Agent, RunContext

agent = Agent('test')

@agent.tool
def foobar(ctx: RunContext[int]) -> int:
    return 123

@agent.tool(retries=2)
async def spam(ctx: RunContext[str]) -> float:
    return 3.14

result = agent.run_sync('foobar', deps=1)
print(result.data)
#> {"foobar":123,"spam":3.14}

```

Parameters:

Name	Type	Description	Default
func	<code>ToolFuncPlain[ToolParams]   None</code>	The tool function to register.	<code>None</code>
retries	<code>int   None</code>	The number of retries to allow for this tool, defaults to the agent's default retries, which defaults to 1.	<code>None</code>
prepare	<code>ToolPrepareFunc[AgentDeps]   None</code>	custom method to prepare the tool definition for each step, return <code>None</code> 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 <code>ToolPrepareFunc</code> .	<code>None</code>

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

624 def tool_plain(
625     self,
626     func: ToolFuncPlain[ToolParams] | None = None,
627     /,
628     *,
629     retries: int | None = None,
630     prepare: ToolPrepareFunc[AgentDeps] | None = None,
631 ) -> Any:
632     """Decorator to register a tool function which DOES NOT take 'RunContext' as an argument.
633
634     Can decorate a sync or async functions.
635
636     The docstring is inspected to extract both the tool description and description of each parameter,
637     [learn more](../agents.md#function-tools-and-schema).
638
639     We can't add overloads for every possible signature of tool, since the return type is a recursive union
640     so the signature of functions decorated with '@agent.tool' is obscured.
641
642     Example:
643     ```py
644     from pydantic_ai import Agent, RunContext
645
646     agent = Agent('test')
647
648     @agent.tool
649     def foobar(ctx: RunContext[int]) -> int:
650         return 123
651
652     @agent.tool(retries=2)
653     async def spam(ctx: RunContext[str]) -> float:
654         return 3.14
655
656     result = agent.run_sync('foobar', deps=1)
657     print(result.data)
658     #> {"foobar":123,"spam":3.14}
659     ...
660
661     Args:
662         func: The tool function to register.
663         retries: The number of retries to allow for this tool, defaults to the agent's default retries,
664             which defaults to 1.
665         prepare: custom method to prepare the tool definition for each step, return 'None' to omit this
666             tool from a given step. This is useful if you want to customise a tool at call time,
667             or omit it completely from a step. See ['ToolPrepareFunc'](pydantic_ai.tools.ToolPrepareFunc).
668     """
669     if func is None:
670
671         def tool_decorator(func_: ToolFuncPlain[ToolParams]) -> ToolFuncPlain[ToolParams]:
672             # noinspection PyTypeChecker
673             self._register_function(func_, False, retries, prepare)
674             return func_
675
676         return tool_decorator
677     else:
678         self._register_function(func, False, retries, prepare)
679         return func
```

result\_validator

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

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

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

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

result_validator(
    func: ResultValidatorFunc[AgentDeps, ResultData]
) -> ResultValidatorFunc[AgentDeps, ResultData]
```

Decorator to register a result validator function.

Optionally takes `RunContext` as its first argument. Can decorate a sync or async functions.

Overloads for every possible signature of `result_validator` are included so the decorator doesn't obscure the type of the function, see `tests/typed_agent.py` for tests.

Example:



```

from pydantic_ai import Agent, ModelRetry, RunContext

agent = Agent('test', deps_type=str)

@agent.result_validator
def result_validator_simple(data: str) -> str:
    if 'wrong' in data:
        raise ModelRetry('wrong response')
    return data

@agent.result_validator
async def result_validator_deps(ctx: RunContext[str], data: str) -> str:
    if ctx.deps in data:
        raise ModelRetry('wrong response')
    return data

result = agent.run_sync('foobar', deps='spam')
print(result.data)
#> success (no tool calls)

```

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

```

503 def result_validator(
504     self, func: _result.ResultValidatorFunc[AgentDeps, ResultData], /
505 ) -> _result.ResultValidatorFunc[AgentDeps, ResultData]:
506     """Decorator to register a result validator function.
507
508     Optionally takes ['RunContext'][pydantic_ai.tools.RunContext] as its first argument.
509     Can decorate a sync or async functions.
510
511     Overloads for every possible signature of 'result_validator' are included so the decorator doesn't obscure
512     the type of the function, see 'tests/typed_agent.py' for tests.
513
514     Example:
515     ```py
516     from pydantic_ai import Agent, ModelRetry, RunContext
517
518     agent = Agent('test', deps_type=str)
519
520     @agent.result_validator
521     def result_validator_simple(data: str) -> str:
522         if 'wrong' in data:
523             raise ModelRetry('wrong response')
524         return data
525
526     @agent.result_validator
527     async def result_validator_deps(ctx: RunContext[str], data: str) -> str:
528         if ctx.deps in data:
529             raise ModelRetry('wrong response')
530         return data
531
532     result = agent.run_sync('foobar', deps='spam')
533     print(result.data)
534     #> success (no tool calls)
535     ...
536     """
537     self._result_validators.append(_result.ResultValidator(func))
538     return func

```