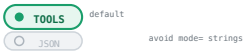
instructor

A look at instructor, which hands back clean, typed Pydantic objects from an LLM instead of raw text, with automatic retries.

Patch a Client

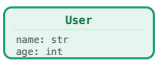
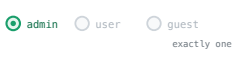
Wrap any OpenAI or Anthropic client; the same instructor surface comes out.

PURPOSE	COMMAND	RESULT
Build + patch from a model string (recommended).	`client = instructor.from_provider ("openai/gpt-5-nano")`	
Same call, an Anthropic model.	`client = instructor.from_provider ("anthropic/claude-4-5-haiku-latest")`	
Patch a client you already configured (OpenAI).	`from openai import OpenAI` `client = instructor.from_openai(OpenAI())`	
Patch an Anthropic client object.	`from anthropic import Anthropic` `client = instructor.from_anthropic(Anthropic())`	
Async client, same shape.	`client = instructor.from_provider ("openai/gpt-5-nano", async_client=True)`	
Pick the structured-output mode.	`instructor.from_provider("…", mode=instructor.Mode.TOOLS)`	

from_provider("provider/model") builds + patches in one call; from_openai / from_anthropic wrap a client you already hold

Define a response_model

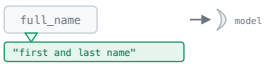
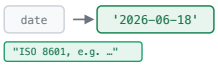
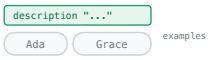
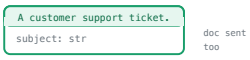
A plain Pydantic model is the schema you want back.

PURPOSE	COMMAND	RESULT
Declare the shape you want.	`class User (BaseModel):` `name: str | age: int`	
Pass it as the response_model.	`client.create(response_model=User,` `messages=[...])`	
The return value is a real instance.	`user.nameis str, user.ageis int`	
Make a field optional.	`nickname: str | None = None`	
Constrain a value.	`age: int = Field(ge=0, le=130)`	
Restrict to a fixed set.	`role: Literal["admin", "user", "guest"]`	

A response_model is plain Pydantic: typed fields, optional with | None, ranges with Field(ge, le), fixed choices with Literal[...]

Field Descriptions Guide the Model

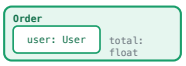
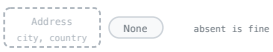
Field(description=...) is prompt engineering inside the schema.

PURPOSE	COMMAND	RESULT
Describe what a field means.	`full_name: str = Field( description="first and last name")`	
Steer the formatting.	`date: str = Field( description="ISO 8601, e.g. 2026-06-18")`	
Constrain and describe together.	`score: int = Field(ge=0, le=100, description="confidence 0-100")`	
Steer with examples.	`Field(description="...", examples=["Ada", "Grace"])`	
The error message guides the re-ask.	`assert "@" in v, "must be an email"` `# inside a @field_validator`	
Whole-model context.	`"""A customer support ticket."""`	

Field(description=...), examples=, constraints, and the model docstring are all sent to steer output, shape it in the schema not the prompt

Nested + list models

Compose models; ask for many at once.

PURPOSE	COMMAND	RESULT
Nest one model inside another.	`class Order(BaseModel): user: User total: float`	
A list of sub-objects.	`items: list[Item]`	
Extract a list as the result.	`client.create( response_model=list[User], ...)`	
Optional nested block.	`address: Address | None = None`	
Deeply typed list value.	`tags: list[Literal["new", "vip"]]`	
Self-documenting nested schema.	`city: str = Field(description=...)`	

Schemas are plain Pydantic, so they compose: nest models, use list[Model] for many objects, and type every element.

Extract One Typed Object

Ragged model text in, a validated object out.

PURPOSE	COMMAND	RESULT
Extract from a sentence.	`user = client.create(response_model=User, messages=[{"role":"user", ...}])`	
Use the OpenAI-shaped call too.	`client.chat.completions.create( response_model=User, messages=[...])`	
Add a system instruction.	`messages=[{"role":"system", "content": "Extract the user."}, {"role":"user", ...}]`	
Anthropic needs max_tokens.	`client.create(response_model=User, max_tokens=1024, messages=[...])`	
Bad input raises an error.	`model returnsage: "old" # raisespydantic.ValidationError`	

create() and chat.completions.create() both accept response_model=; you get back a validated User, not a string to parse

Automatic retries

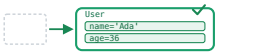
A failed validation is re-asked with the error, not crashed.

PURPOSE	COMMAND	RESULT
Validation failure triggers a retry.	`client.create(response_model=User, max_retries=3, messages=[...])`	
A custom rule that can fail.	`@field_validator("name") def caps(cls, v): assert v.istitle()`	
The model sees the error message.	`"name must be Title Case" -> model`	
Fine-grained retry control (tenacity).	`from tenacity import Retrying, stop_after_attempt max_retries=Retrying(stop=stop_after_attempt(3))`	
Give up after the budget.	`retries exhausted -> InstructorRetryException`	
Omit it: one retry by default.	`client.create(response_model=User, messages=[...])`	

max_retries feeds the validation error back to the model and re-asks; a tenacity Retrying(...) gives full control, InstructorRetryException raises when the budget runs out

Stream Partial Objects

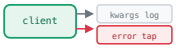
Watch fields fill in live instead of waiting for the whole object.

PURPOSE	COMMAND	RESULT
Stream one object as it fills.	`stream = client.create_partial( response_model=User, stream=True, ...)`	
Consume the partial stream.	`for partial in stream: print(partial)`	
Stream many objects (iterable).	`for user in client.create_iterable( response_model=User, messages=[...]): ...`	
Async streaming.	`async for partial in stream: ...`	
Literal fields need the mixin.	`class User(BaseModel, PartialLiteralMixin):`	
Final frame is a normal object.	`# last yielded value is a complete User`	

create_partial(stream=True) yields one object filling in; create_iterable streams many; the last value is a complete, validated object

with_completion (raw access)

Get the typed object AND the raw response together.

PURPOSE	COMMAND	RESULT
Get object + raw completion.	`user, completion = client.create_with_completion( response_model=User, messages=[...])`	
Read token usage.	`completion.usage`	
Reach the raw model id / metadata.	`completion.model, completion.id`	
Inspect what the model sent.	`completion.choices[0].message`	
Hook into every call (logging).	`client.on("completion:kwargs", log_fn) client.on("completion:error", err_fn)`	
The object is unchanged.	`user ==create(...) result`	

create_with_completion returns (object, completion); read completion.usage / .model / .id for the raw SDK response, hook calls with client.on(...)