08

Type Hints & Pydantic

Describe your data, then make Python enforce it

Chapter 8 · the reliability superpower behind every agent codebase

08 The journey — from hints to a runtime gate

You'll meet the same shape in every agent codebase: a class that describes the data, handed to an LLM, then used to CHECK whatever comes back. Here are the five steps that build up to it.

Hints are just notes the runtime ignores. Pydantic is the tool that finally makes them bite — rejecting bad data at the door.

08 Type hints — notes about your data

Python checks types as it runs, not up front

A type hint is a NOTE saying what kind of value something should be (the words are 'annotation' or 'type hint'). You can add them, tools read them — but the interpreter happily ignores them. That mix is called 'gradual typing'.

```
model_name: str = 'claude-opus-4'
temperature: float = 0.7
max_tokens: int | None = None      # an int, or nothing
(None)

def truncate(text: str, limit: int = 100) -> str:
    return text[:limit]              # the -> str is the
RETURN type
```

JS to TS to Python

This is TypeScript's syntax rearranged.

const n: string becomes n: str

function f(a): string becomes
def f(a) –> str:

The –> str is the : string TS puts in the same spot.

08 The catch — hints don't stop anything

TypeScript has a gate. Python has none.

TypeScript forces your code through a compiler that REFUSES to build broken code. Python has no such gate — you can run code that breaks every annotation it declares.

```
def add(a: int, b: int) -> int:
    return a + b

print(add('no', 'check'))    # => nocheck    <- runs fine!
```

The hint is a lie the runtime won't catch.

Hold onto this — it is the entire reason Pydantic exists.

08 The type words you'll read constantly

Modern Python uses the plain container names directly

```
tags: list[str]                 # a list of strings
scores: dict[str, int]          # str keys, int values
point: tuple[float, float]      # a fixed pair of decimals
retries: int | None             # an int, OR nothing (None)
token_id: int | str             # one of several types
role: Literal['system', 'user', 'assistant']    # only these
formatter: Callable[[int], str]    # takes an int, returns a
str
payload: Any                    # 'switch checking OFF here'
```

JS to TS to Python

`string[]`	`list[str]`
`Record<s,n>`	`dict[str,int]`
`X | undefined`	`X | None`
`A | B`	`A | B` (same!)
`any`	`Any`
`'a' | 'b'`	`Literal['a','b']`

Literal is TS's string union — it constrains LLM roles and tool names everywhere.

'Any' means typing was switched off for this value. One Any at a real boundary is fine — Any everywhere is a smell: the safety net is gone.

08

Three more you'll spot

Naming shapes, naming types, attaching metadata

TypedDict — describe the shape of a plain dict (a JS-object-like thing) that STAYS a dict, but you want editor help on its keys.

Type alias — give a complex type a short name. Annotated[T, ...] — keep the type T but attach extra info a tool can read. Pydantic and FastAPI mine this heavily.

```
class Message(TypedDict):
    role: Literal['user', 'assistant']
    content: str
msg: Message = {'role': 'user', 'content': 'Hi'}   # still a normal dict

type Embedding = list[float]              # 3.12 alias: Embedding == list[float]
PositiveInt = Annotated[int, 'must be > 0']   # the string is metadata
```

08 Static checkers — Python's stand-in compiler

mypy and pyright read the hints, so the runtime doesn't have to

Because the runtime ignores hints, a SEPARATE tool reads them and flags contradictions. Two common ones: mypy and pyright (pyright powers VS Code's Pylance, so you may already run it). They catch the same bugs TypeScript catches.

```
def greet(name: str) -> str:
    return 'Hi ' + name

greet(42)    # mypy: incompatible type 'int'; expected 'str'

result = lib.call()  # type: ignore[no-untyped-call]  #
narrow, justified
```

Reading and judging

A file dotted with bare `# type: ignore` (no code, no reason) is a red flag — each one silenced the checker instead of fixing the type.

Prefer `# type: ignore[code]` so only the named error is muted.

08 Pydantic v2 — the runtime gate

Declare a model, and it ENFORCES the types on real data

Pydantic is zod. A zod schema validates unknown JSON at runtime and hands you a typed value — a Pydantic BaseModel does exactly the same. The schema IS the type, written once.

```
class ChatRequest(BaseModel):
    model: str                          # required, must be str
    messages: list[str]                 # required list of str
    temperature: float = 0.7            # optional — has a default
    max_tokens: int | None = None       # optional, may be None
    top_p: float = Field(default=1.0, ge=0.0, le=1.0)  # constraints

req = ChatRequest(model='claude-opus-4', messages=['hello'])
```

Required vs optional is decided by ONE thing: does the field have a default? No default = required. Field() adds limits like ge (>=) and le (<=).

08

Coercion, and the error when it can't fit

It converts what it sensibly can, and rejects what it can't

```
r = ChatRequest(model='x', messages=['hi'], temperature='0.9')
print(r.temperature, type(r.temperature))
# => 0.9 <class 'float'>   <- the string was COERCED to a float
```

```
try:
    ChatRequest(model='x', messages='not-a-list', top_p=5)
except ValidationError as e:
    print(e.error_count())          # => 2
    print(e.errors()[0]['loc'])     # => ('messages',)
    # messages must be a list; top_p=5 breaks the le=1.0
limit
```

Coercion surprises strict-TS people: Pydantic will turn '0.9' into 0.9.

When data truly can't fit, it raises ValidationError with a precise report — loc (where), type (what rule), msg (why).

That report is gold for logging or feeding back to the LLM.

08

Four methods everywhere — and a version trap

Every v2 method starts with model_

```
# FROM data -> validated model
ChatRequest.model_validate({'model': 'x', 'messages': ['hi']})
ChatRequest.model_validate_json('{...json string...}')

# FROM model -> data
req.model_dump()         # => a dict
req.model_dump_json()    # => a JSON string
```

Reading and judging — the #1 version trap

Pydantic v1 used .dict() .json() parse_obj() @validator and an inner class Config.

Pydantic v2 (everything current) renamed them to .model_dump() .model_dump_json() .model_validate() @field_validator and model_config = ConfigDict(...).

See .dict() or class Config: or @validator ? You're reading v1-era code — it will break or warn on a modern install. Flag it.

08 Nested models, validators, computed fields

Where the real checking logic lives

```
class Customer(BaseModel):
    model_config = ConfigDict(extra='forbid')  # no unknown keys
    name: str
    email: str
    address: Address                # NESTED model, checked
recursively
    orders: list[float] = Field(default_factory=list)    # mutable

    @field_validator('email')       # checks/normalizes ONE field
    @classmethod
    def email_has_at(cls, v: str) -> str:
        if '@' not in v: raise ValueError('need @')
        return v.lower()            # returned value REPLACES the
field

    @computed_field                 # derived, shown in model_dump()
    @property
    def order_total(self) -> float:
        return sum(self.orders)
```

Rules to remember

@field_validator sits above @classmethod and sees ONE field's value.

@model_validator(mode='after') runs on the finished object — that's where CROSS-field checks go.

@computed_field sits above @property and, unlike a plain property, appears in model_dump().

Use default_factory=list, never default=[] — a bare [] is shared across every instance (a real footgun).

08

Loading config from the environment

The standard way GenAI apps read API keys

pydantic-settings (a separate install) reads config from environment variables and .env files straight into a validated model. A missing required key fails LOUDLY at startup — not three calls deep.

```
class Settings(BaseSettings):
    model_config = SettingsConfigDict(env_file='.env', env_prefix='APP_')

    anthropic_api_key: str              # required -> APP_ANTHROPIC_API_KEY
    model: str = 'claude-opus-4'         # APP_MODEL, default provided
    max_retries: int = Field(default=3, ge=0)   # coerced from the env string

settings = Settings()   # reads real env vars / .env at call time
```

Better an instant, clear crash at boot than an 'undefined API key' error surfacing deep inside a request.

08

Why this is THE GenAI reliability pattern

LLMs return text — so validate it at the door

Even in 'JSON mode' an LLM can hallucinate a field, drop a required one, or send a number as a string. The defense: define a Pydantic model as your target, send its JSON schema to the model, then validate whatever comes back.

```
GOOD response — validates cleanly

currency: 'GBP'   (3 chars, ok)
amount: 40.0   (must be >= 0, ok)

model_validate_json(good)
-> accepted, flows on safely
```

```
BAD response — hallucinated, REJECTED

currency: 'POUNDS'   (not 3 chars)
amount: -5   (must be >= 0)

except ValidationError as e:
  e.errors() -> loc tells you EXACTLY
  which fields failed — caught before
  it ever hit your database.
```

Malformed output is rejected at the boundary instead of poisoning your program.

08 Guarding agent actions — the #1 thing to flag

An agent's next step is untrusted LLM text

```
class AgentAction(BaseModel):
    tool: Literal['search', 'calculator', 'send_email']  # only these
    args: dict[str, str]

# If the LLM invents tool='delete_database', model_validate
# raises ValidationError — the illegal action never runs.
```

The finding to hunt

```
data = json.loads(llm_output)
do_thing(data['tool'])    # <- untrusted text,
straight into code
```

A missing key raises KeyError mid-flow; a hallucinated tool or amount flows unchecked into execution.

The correct shape

```
Model.model_validate_json(llm_output)
at the boundary, wrapped in
try / except ValidationError
with a retry or fallback.
```

Trace every LLM response to its first use. No gate between model and action = the finding.

08 Pick the right tool — and the big lesson

Tool	Runtime check?	Use it for
`dataclass`	No (hints only)	Trusted internal data you build yourself — config, results
`Pydantic BaseModel`	YES	Any data crossing a BOUNDARY — LLM output, APIs, env vars
`TypedDict`	No (hints only)	Data that must stay a plain dict, but you want key hints

The frontend rule transfers exactly: use interfaces (dataclass / TypedDict) for data you already trust inside your own code; reach for zod / Pydantic the moment data comes from OUTSIDE — and in GenAI work, the LLM is always outside.

The teaching — hints describe your data, but only Pydantic makes Python enforce it. Put a validation gate between the LLM and your code, and malformed output dies at the door instead of inside your program.