

Type Hints in Real-World Projects: Practical Steps for Continuous Maintenance and Improvement

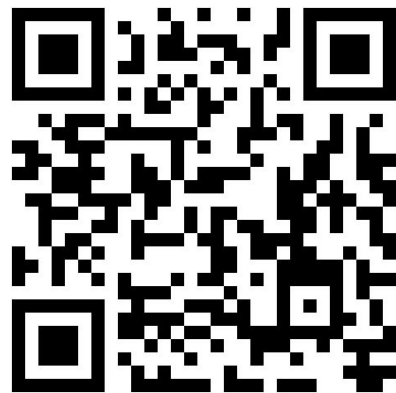
Koudai Aono

Goal

- **Typing as a maintenance tool, not a one time task**
- **Incremental workflow anyone can adopt**
- **Leave with a ready checklist**

About Me

- OSS developer at Mirascope
- My OSS projects:
 - Developing PyCharm plugins for Pydantic and Ruff, “datamodel-code-generator”
- GitHub:
<https://github.com/koxudaxi>
- PEP 750 co-author
 - Template strings 🎉 - Python 3.14
- Speaker @ PyCon US 2024, 2025 / EuroPython 2024



How I Came Up With This Talk

- 2024 decorator talk → many questions on keeping types healthy
- Real projects drift: `typing.Any`, legacy syntax, blanket `# type: ignore`
- Need practical, incremental workflow

1. Intro Example

Intro Example — We work at a SaaS Company

1. Create new product with type-hint
2. v1.0 - 0 × `typing.Any`, 0 × `# type: ignore` ✓
3. Sales persons requests a lot of new features.
4. We don't have the time to resolve type-puzzle 😭
5. v2.0 200 × `typing.Any`, 50 × `# type: ignore` 😱
6. Static checker stays silent
→ bugs slip into prod 🐛

v1.0 Strict Code (Good)

```
from typing import TypedDict
```

```
class Item(TypedDict):  
    id: int  
    price: int
```

PASSED



```
def total(items: list[Item]) → int | float:  
    # some logic...
```

```
$ pyright v1.py
```

v2.0 Any Festival 🤡

```
from typing import Any
```

```
def total(items: dict) → Any:  
    ...
```

PASSED



unseen bugs



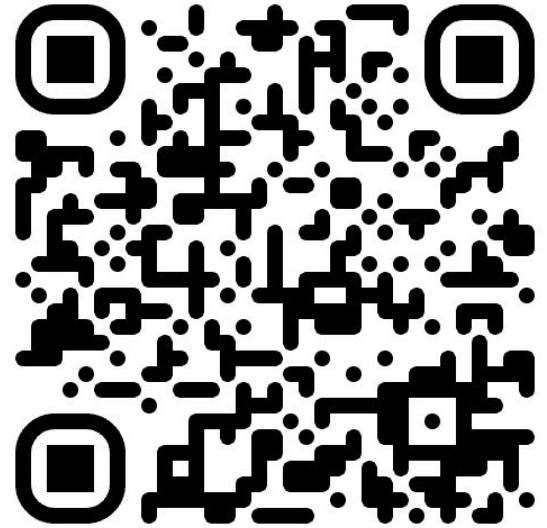
```
$ pyright v2.py
```


1. Recap: Intro Example

- Typing is like tests—if you tape over warnings, you have no tests.
- We should NOT ignore the type-warning

Agenda

1. Intro Example
2. Newer feature ideas
3. pyupgrade for modernizing
4. Partial ignores & `pre-commit`/CI
5. Real Mirascope case
6. Gradual adoption
7. Conclusion



Code repo URL of Talk

https://github.com/koxudaxi/europython_2025

2. Newer feature ideas

New Typing Features

- `typing.TypeGuard[T]`
user-defined narrowing predicate (PEP 647 / Py 3.10)
- `typing.TypeIs[T]`
checker-only “it is T” predicate (PEP 742 / Py 3.13)
- `typing_extensions`
Both ship today via `typing_extensions` (works on 3.9+)

typing.TypeGuard can't narrow the else branch

```
from typing import TypeGuard
```



```
def is_int(x: str | int) → TypeGuard[int]:  
    return isinstance(x, int)
```

```
def use(x: str | int) → str | int:  
    if is_int(x):  
        return x + 1          # ✓ int  
    else:  
        return x.upper()     # ⚠ may still be int
```

typing.TypeIs narrows both branches

```
from typing import TypeIs
```

```
def is_int(x: str | int) → TypeIs[int]:  
    return isinstance(x, int)
```

```
def use(x: str | int) → str | int:  
    if is_int(x):  
        return x + 1          #  int  
    else:  
        return x.upper()     #  str
```




“Upgrade-Wait” Syndrome 🤔

“TypeIs looks great, but our codebase is still on Python 3.12.

We can’t use the feature until we bump to 3.13...”

- New Python versions drop every 12 months (PEP 602)

Why it Hurts

- Upgrading Python is costly \Rightarrow teams keep postponing
- Every release needs compat-check across libs & CI before rollout
- Consequences:
 -  slow adoption
 -  stale hints
 -  higher maintenance cost

Solution: typing_extensions

```
pip install --upgrade typing_extensions
```

```
# runs on 3.9+
```

```
from typing_extensions import TypeIs
```



- Official back-port package
(maintained by the CPython team)
- Pin once in CI → whole team gets the same feature set

2. Recap: Intro Example

- `typing_extensions` = an official Time-machine 🚗💨🔥🔥⚡🕒
- Ship tomorrow's typing features today (e.g. `typing.TypeIs` on 3.9+).

3. pyupgrade

Out-of-Date Hints

- Legacy: `typing.List`, `typing.Dict`, `Optional[int]` clutter code
- Current: `list`, `dict[int]`, `int | None`
- Manual cleanup is slow & error-prone
- Result:
 -  mixed styles
 -  extra review time

Meet pyupgrade (stand-alone CLI)

```
pip install pyupgrade  
pyupgrade --py311-plus **/*.py
```

- One-shot converter by @asottile
- Rewrites:
 - `typing.List[int]` → `list[int]`
 - `Optional[T]` → `T | None`, etc.
- Zero config: run once, commit the diff, done

pyupgrade in Action ⚡

Before

```
from typing import List,  
Dict, Optional, Union
```

```
def load(  
    data: Union[str,  
bytes]  
) →  
Optional[List[int]]:  
    ...
```

After

```
from typing import List,  
Dict, Optional, Union
```

```
def load(  
    data: str | bytes  
) → list[int] | None:  
    ...
```

pyupgrade in Action ⚡

Before

```
from typing import List,  
Dict, Optional, Union
```

```
def load(  
    data: Union[str,  
bytes]  
) →  
Optional[  
    ...
```

After

```
from typing import List,  
Dict, Optional, Union
```

```
def load(  
    data: str | bytes  
) → list[int] | None:
```

Unused types stay here 😬

One-Command Cleanup with Ruff 🚀

```
ruff check --fix --select UP,F401 .
```

- UP → all pyupgrade transforms (modernizes types & syntax)
- F401 → removes unused imports
- Adds auto-fixes, runs fast ⇒ cleaner codebase in seconds

Before → After — Ruff UP Cleanup

Before

```
from typing import List,  
Dict, Optional, Union  
  
def load(  
    data: Union[str,  
bytes]  
) →  
Optional[List[int]]:  
    ...
```

After

```
def load(  
    data: str | bytes  
) → list[int] | None:  
    ...
```

3. Recap: pyupgrade + Ruff

- pyupgrade: one-shot legacy cleanup
- Ruff UP,F401: auto-modern on every commit
- Result: consistent, future-proof typing with near-zero effort

4. Partial ignores & pre-commit/CI

“Magic” is not illegal in Python

```
class AppClient: ...  
client = AppClient()  
def get_name(user_id: str) → str: ...  
client.name = get_name("123")
```

```
$ mypy .  
... error: "AppClient" has no attribute "name"  
[attr-defined]
```

“Quick Fix with # type: ignore”

```
class AppClient: ...  
client = AppClient()  
def get_name(user_id: str) → str: ...  
client.name = get_name("123") # type: ignore
```

PASSED



- Only goal: silence the attr-defined error

“Quick Fix with `# type: ignore`” → Blanket Ignore 🤪

```
class AppClient: ...  
client = AppClient()  
def get_name(user_id: uuid.UUID) → str: ...  
client.name = get_name("123") # type: ignore
```

PASSED ✅ - unseen bugs 🐛💣

- Only goal: silence the lint defined error
- CI turns green... but the line now hides all future issues

“Quick Fix with specific error”

```
def get_name(user_id: UUID) → str: ...
```

```
client.name=get_name("123")#type:ignore[attr-defined]
```

Failed  - Found bugs  

```
$ mypy .
```

```
... error: Argument 1 to "get_name" has  
incompatible type "str"; expected "UUID" [arg-type]
```

“Quick Fix with specific error”

```
def get_name(user_id: UUID) → str: ...  
user_id: uuid.UUID = ...  
client.name=get_name(user_id)#type:ignore[attr-defined]
```

PASSED



- Only attr-defined is muted; arg-type remains checked
- Future signature changes surface in CI, preventing hidden crashes

Pre-commit: pyright Instant Check

```
# .pre-commit-config.yaml
- repo: https://github.com/pre-commit/mirrors-pyright
  rev: v1.1.400
  hooks:
    - id: pyright
      args: ["--python-version", "3.13"]
```

- Reads pyproject.toml → no duplicate config
- Runs on every commit

Same Check in CI

```
- name: Type check  
  run: pyright --python-version 3.12
```

```
# pyproject.toml (single source of truth)  
[tool.pyright]  
typeCheckingMode = "strict"  
reportUnusedTypeIgnoreComment = "error"
```

- Same pyproject in local & CI → always in sync
- Single file → consistent results, zero drift

4. Recap: Precise Ignores + Automated Checks

- Use specific error codes (`attr-defined`, `arg-type`, ...)
 - one bug silenced, others still visible
- Store Pylint settings in one `pyproject.toml`
 - same rules in pre-commit and CI
- Result: early detection, no hidden drift, safer merges

5. Real Mirascope case

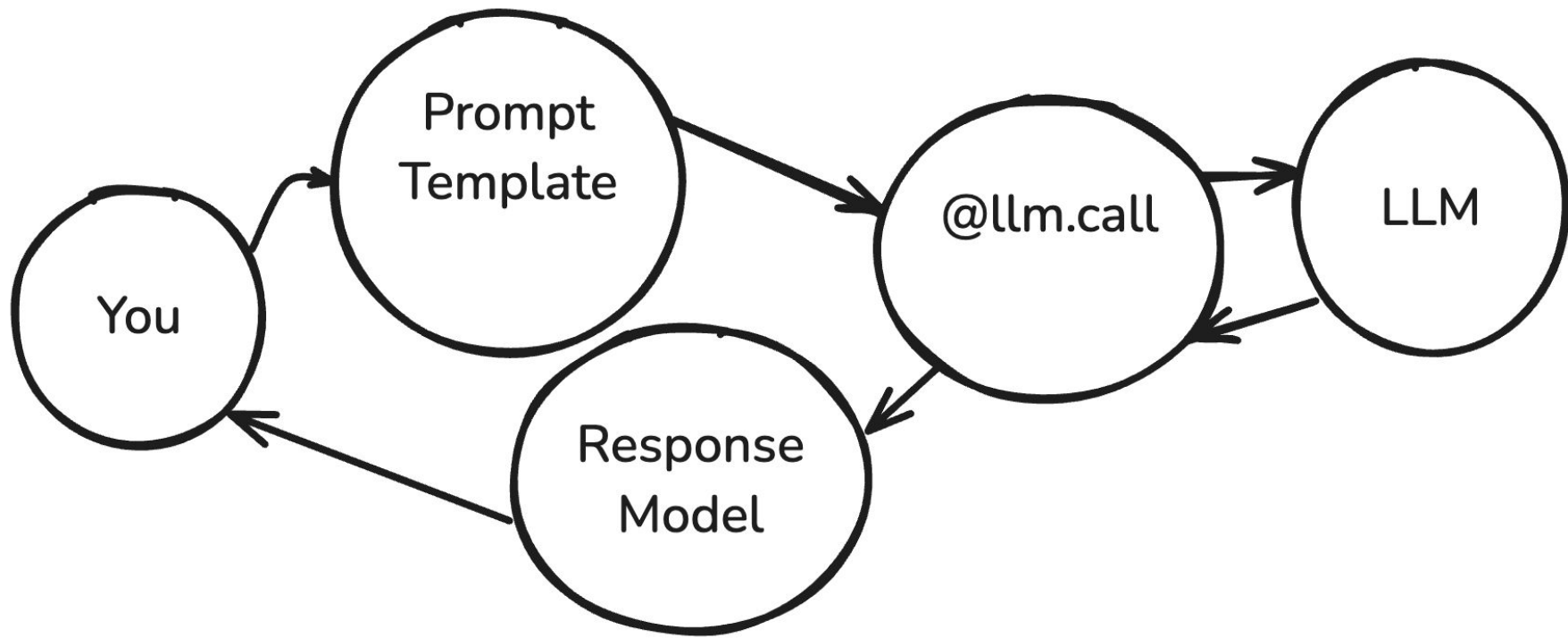
What is Mirascope? 🐸

- A single Python SDK that talks to many LLMs (OpenAI, Anthropic, Gemini, ...)
- Simple `@llm.call(...)` decorator, sync & async, with strong type hints
- Install + key → start generating in seconds
(`pip install "mirascope[openai]"`)

2024 EuroPython Talk Recap — Typed Decorators

- `typing.ParamSpec` / `typing.Concatenate` → decorator can rewrite a function's signature without losing type safety
- The same trick powers today's `@llm.call` (adds `response_model`, etc.)

How to work @llm.call(...) decorator



```
from mirascope import llm
from pydantic import BaseModel
```

```
class Book(BaseModel):
    title: str
    author: str
```

```
@llm.call(provider="openai", model="gpt-4o-mini",
           response_model=Book,)
```

```
def extract_book(text: str) → str:
    return f"Extract the book: {text}"
```

```
text = "The Name of the Wind by Patrick Rothfuss"
book: Book = extract_book(text)
```


Examples Are Type-Checked

- **examples/ directory: multiple real-usage scripts committed with the repo**
- **pyright runs on src + examples in both pre-commit and CI**
 - **type errors caught before any live API call**

Pyright Caught My Bug During call_params Refactor

- I was adding a new call_params path inside the `@llm.call` decorator
- An example script failed pyright, showing my mistake before merge
- The code on this slide is a greatly simplified excerpt of that real (and messy) task

Call Params Bug 🛠️ (real my mistake, greatly simplified)

```
from mirascope import llm

@llm.call(
    provider="openai",
    model="gpt-4o-mini",
    call_params={"temperature": 0.7}
)
def recommend_book(genre: str) → str:
    return f"Recommend a {genre} book"
```

The definition of CallParam type

```
from typing import TypedDict, NotRequired

class BaseCallParams(TypedDict, total=False):
    ...




class OpenAICallParams(BaseCallParams):
    temperature: NotRequired[float | None]
```

I made mistake 😬 Change existing type to incorrect type

```
from typing import TypedDict, NotRequired
```

```
class BaseCallParams(TypedDict, total=False):
```

```
...
```

 float → str  

```
class OpenAICallParams(BaseCallParams):
```

```
    temperature: NotRequired[str | None]
```

pre-commit/CI Flags It Immediately 🚨

```
"temperature" has incompatible type "float";  
expected "str | None" [typeddict-item]
```

```
# quick fix in the same PR
```

```
temperature: NotRequired[float | None] # ✅
```

- Type-checked example script caught the issue—no live API calls required.

Recap: Why Typed Examples Matter 🌍

- Example scripts + TypedDict / Pydantic = lightweight interface tests
- CI type-checks catch API mismatches before runtime
- Scales across many external SDKs & services — fast, no heavy mocks

Gradual Adoption Roadmap

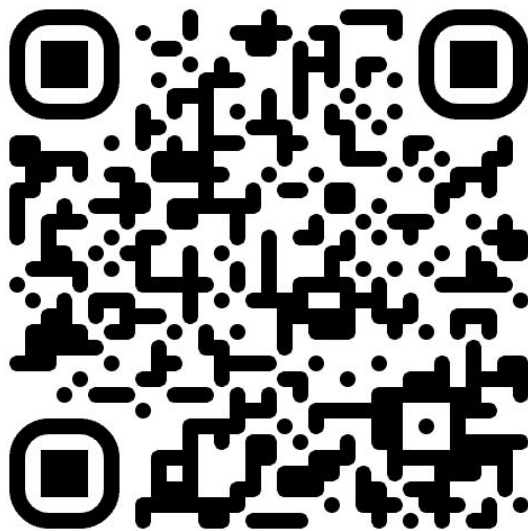
1. Ruff UP,F401 — auto-fix + hook
2. mypy / pyright (strict) — modern type check
3. Precise ignores — drop blanket tags
4. typing_extensions — new types on old Python
5. Typed examples — fast interface tests
6. Gradual rollout (one module at a time)

Conclusion

- Typing = reliable foundation
 - fewer bugs
 - safer refactors
 - smoother upgrades
- Maintain hints little-by-little → big pay-off over time

Thank you very much!!

Here is the URL of Repository in the Talk



`https://github.com/koxudaxi/europython_2025`