infobip

Schema Beats Prompt

Designing Trustworthy AI APIs

Presented at ISCon Banja Luka 2026

Edvin Teskeredzic | Senior AI Software Engineer

MARCH 2026

What we'll talk about

1. What "trustworthy" means for an API
2. FastAPI + Pydantic: Contracts as the Default
3. Breaking stuff with AI
4. Enforcing structure on LLMs
5. Designing usable AI APIs
6. How this affects DevEx
7. Make AI APIs boring again

About me

- Senior AI Software Engineer @ Infobip

About me

- Senior AI Software Engineer @ Infobip
- Published some research papers 📚

About me

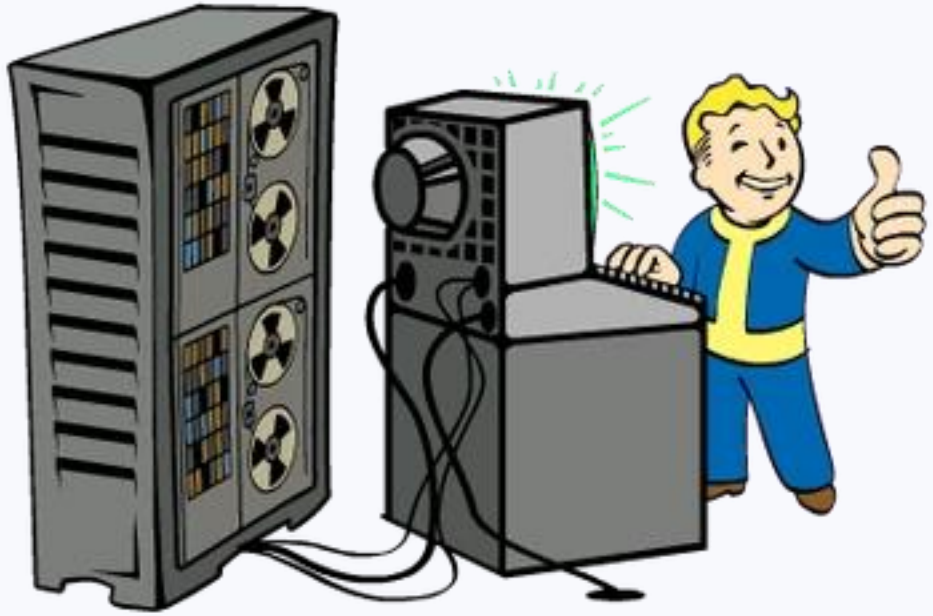
- Senior AI Software Engineer @ Infobip
- Published some research papers 📚
- Some of my (nerdy) hobbies:
  - Pub quizzes
  - Sci-Fi and Fantasy
  - Gaming

What I do at work

- AI and ML (obviously)

... but also...

- Developer Experience (DevEx)
- Systems Design and ML infra
- Tooling, observability, traceability
- API design

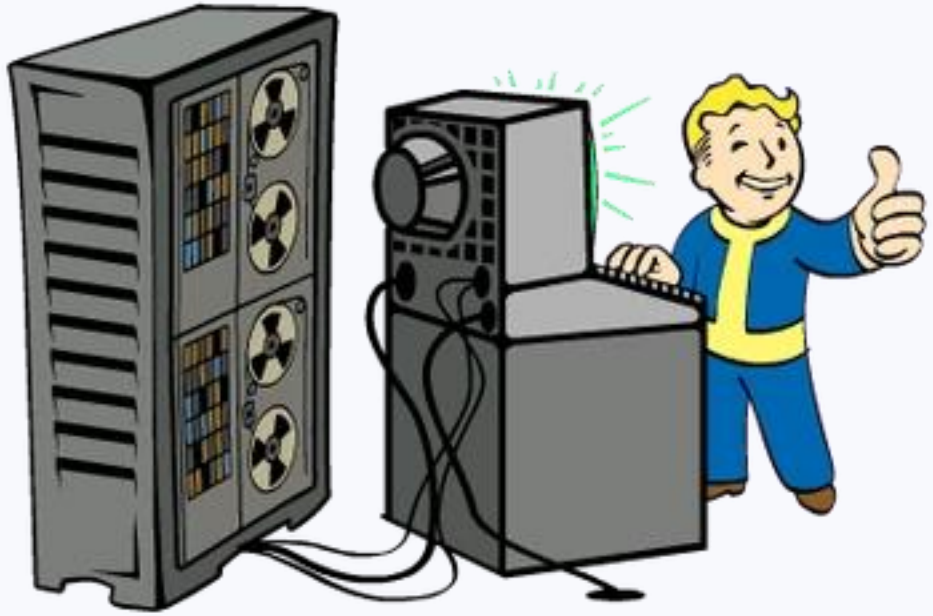


What I do at work

- AI and ML (obviously)

… but also…

- Developer Experience (DevEx)
- Systems Design and ML infra
- Tooling, observability, traceability
- **API design**



We'll be talking about this today!

Describe an API in three words

- **Deterministic (regarding their contract)**
- **Explicit**
- **Predictable**

Describe an API in three words

- **Deterministic (regarding their contract)**
- **Explicit**
- **Predictable**

Now, imagine we let a little kid play with our API responses before sending them back to the user...

Describe an API in three words

- **Deterministic (regarding their contract)**
- **Explicit**
- **Predictable**

Now, imagine we let a little kid play with our API responses before sending them back to the user...

...this happens when we introduce AI (LLMs) to our API

AI APIs…

- Are **probabilistic**
- Have **best-effort outputs**
- Fail **creatively**

AI APIs…

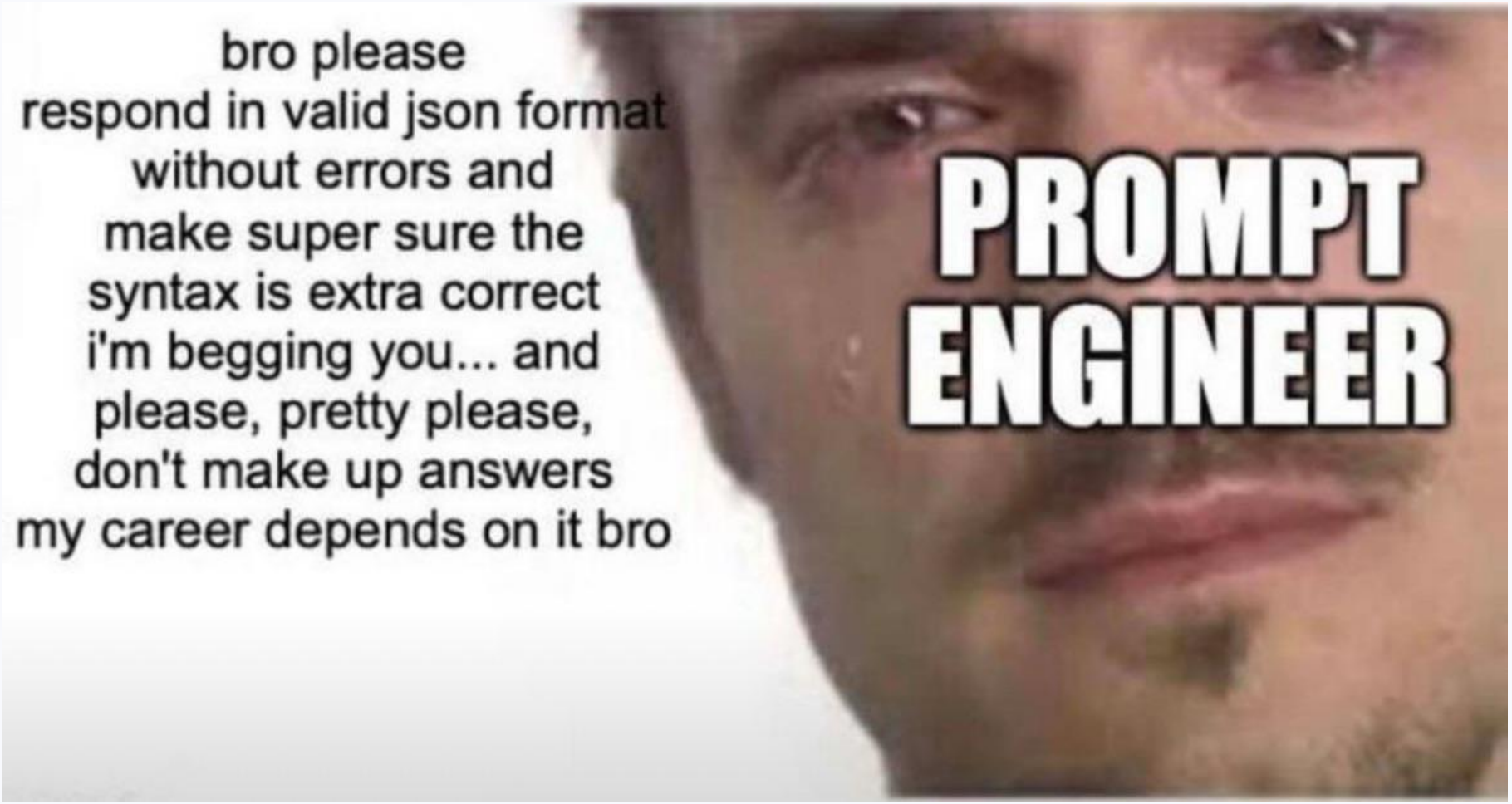
- Are **probabilistic**
- Have **best-effort outputs**
- Fail **creatively**

When a traditional API fails, it's explicit.
When an LLM fails, it **confidently makes something up – implicit failure**!

A simple example

An LLM could do something like:

- `"confidence": "high"`
- Missing **`summary`** field
- Additional fields we did not ask for
- Invalid JSON…


bro please
respond in valid json format
without errors and
make super sure the
syntax is extra correct
i'm begging you... and
please, pretty please,
don't make up answers
my career depends on it bro
PROMPT
ENGINEER

So, what's the issue?

- Is this a model problem?
  - Nope.
- Oh, so a prompt problem then?
  - No, try again.

So, what's the issue?

- Is this a model problem?
  - Nope.
- Oh, so a prompt problem then?
  - No, try again.

TAKEAWAY

Prompting helps the model **understand intent.**

But **schemas are what make APIs trustworthy!**

What we'll show today

Show how **the Python ecosystem** lets us enforce structure *even when the thing generating the data is non-deterministic!*

Remember: The goal isn't smarter prompts.

The goal is making AI APIs boring again!

What “trustworthy” means for an API

What does it mean to trust an API?

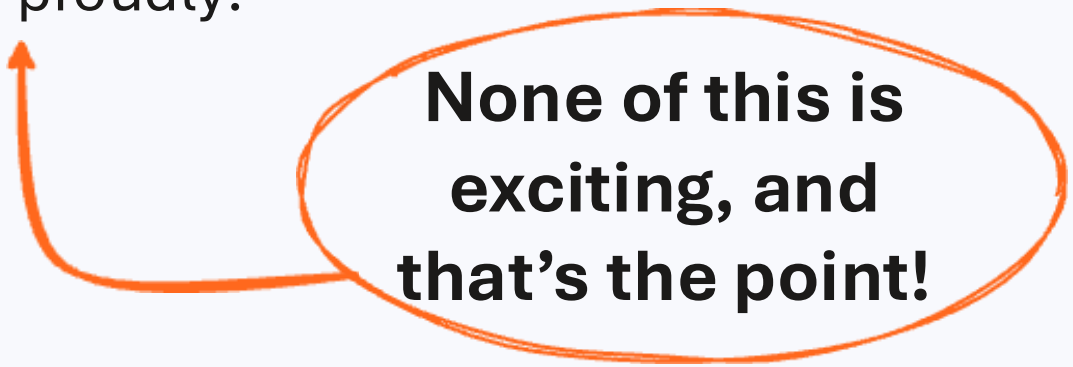
A trustworthy API does a few **boring** things extremely well:

1. It rejects invalid input
2. It never returns invalid output (we rather have it fail!)
3. Stable + explicit contract
4. Fail loudly and proudly!

What does it mean to trust an API?

A trustworthy API does a few **boring** things extremely well:

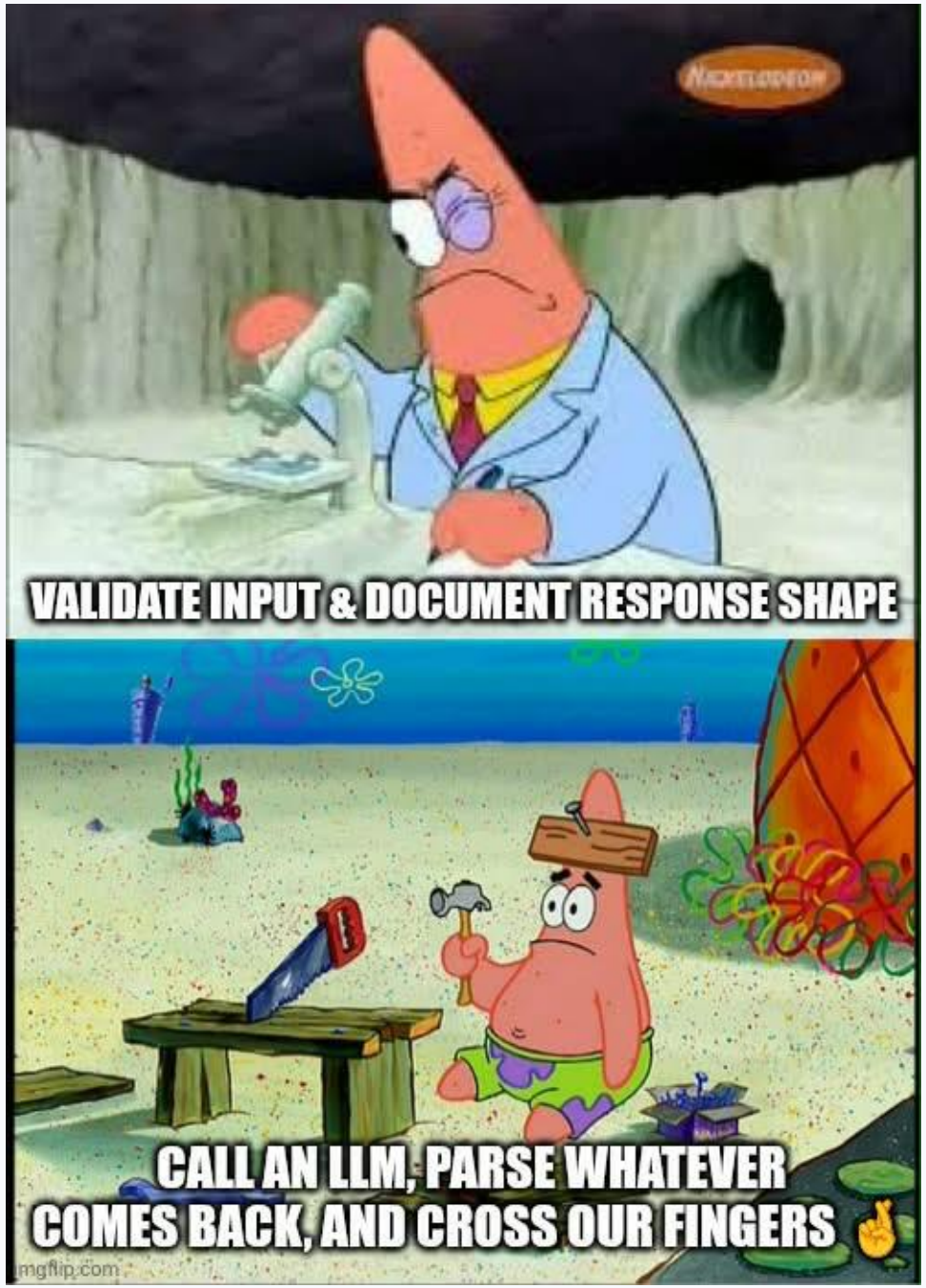
1. It rejects invalid input
2. It never returns invalid output (we rather have it fail!)
3. Stable + explicit contract
4. Fail loudly and proudly!



We know how to be boring

1. Request schemas
2. Response schemas
3. Validation at the boundary
4. Clear error codes

But something strange happens when we add LLMs to the mix...



We treat an unpredictable component as trusted!

THIS IS ABOUT CONTRACT ENFORCEMENT

If your API contract stops at the LLM boundary, you don’t really have a contract.

Let’s be boring

The question becomes: **How do we enforce the same boring guarantees (validation, schemas, explicit failures) when the data is generated by an AI?**

Let’s be boring

The question becomes: **How do we enforce the same boring guarantees (validation, schemas, explicit failures) when the data is generated by an AI?**

And this is where **gets interesting!**

FastAPI + Pydantic: Contracts as the Default

Refresher for non-Python people

- **FastAPI** – High-performance web framework
- **Pydantic** – Data validation with Python models

Python is great for API design

In **FastAPI**, you do not describe an API in three places:

1. You don’t define a request schema
2. You don’t define validation logic
3. You don’t define documentation

You define one thing – a Pydantic model.

Pydantic model example

```python
class UserReq(BaseModel):
    """Create user request."""
    email: EmailStr = Field(..., description="Valid email")
    name: str = Field(..., min_length=2, description="2+ chars")

    @field_validator("name")
    @classmethod
    def strip_name(cls, v: str) -> str:
        v = v.strip()
        if not v: raise ValueError("name required")
        return v
```

Pydantic model example

```python
class UserReq(BaseModel):
    """Create user request."""
    email: EmailStr = Field(..., description="Valid email")
    name: str = Field(..., min_length=2, description="2+ chars")

    @field_validator("name")
    @classmethod
    def strip_name(cls, v: str) -> str:
        v = v.strip()
        if not v: raise ValueError("name required")
        return v
```

Includes:

- **Request schema**

Pydantic model example

```python
class UserReq(BaseModel):
    """Create user request."""
    email: EmailStr = Field(..., description="Valid email")
    name: str = Field(..., min_length=2, description="2+ chars")

    @field_validator("name")
    @classmethod
    def strip_name(cls, v: str) -> str:
        v = v.strip()
        if not v: raise ValueError("name required")
        return v
```

Includes:

- **Request schema**
- **Documentation**

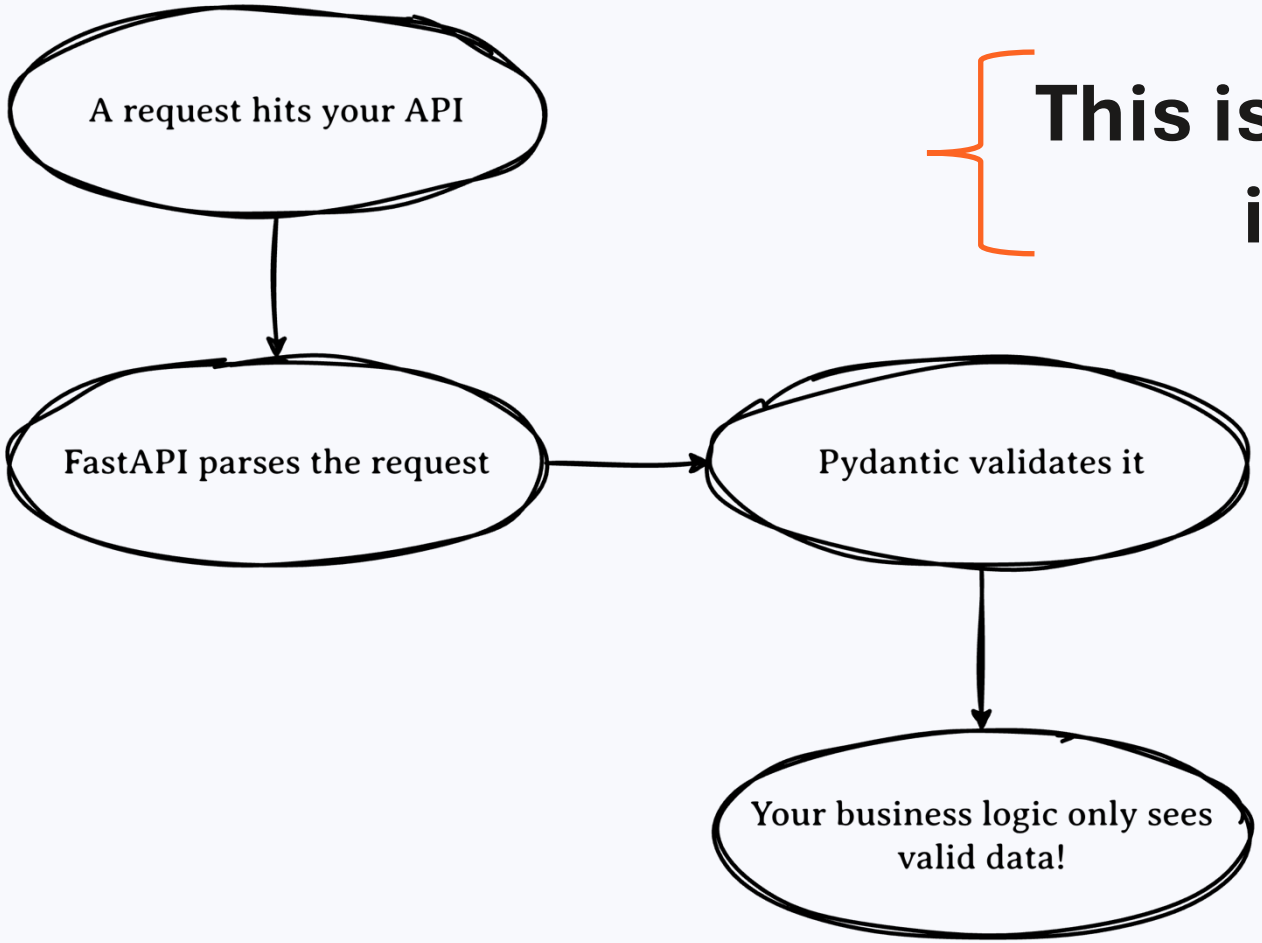
Pydantic model example

```python
class UserReq(BaseModel):
    """Create user request."""
    email: EmailStr = Field(..., description="Valid email")
    name: str = Field(..., min_length=2, description="2+ chars")

    @field_validator("name")
    @classmethod
    def strip_name(cls, v: str) -> str:
        v = v.strip()
        if not v: raise ValueError("name required")
        return v
```

Includes:

- **Request schema**
- **Documentation**
- **Validation**

This is how we do



This is not optional, this is the default

But wait, there’s more!

- Docs are generated automatically – the same model becomes your OpenAPI schema
- Clients don’t have to guess
- SDKs can be generated safely

If it’s not in the schema, it doesn’t exist

You thought it was over? Think again!

- Most people think validation stops at input – it does not.
- FastAPI + Pydantic can validate **responses**
- Outgoing data is checked against the declared schema
- Extra fields are dropped, missing fields raise errors

Your API cannot lie.
Not even accidentally.

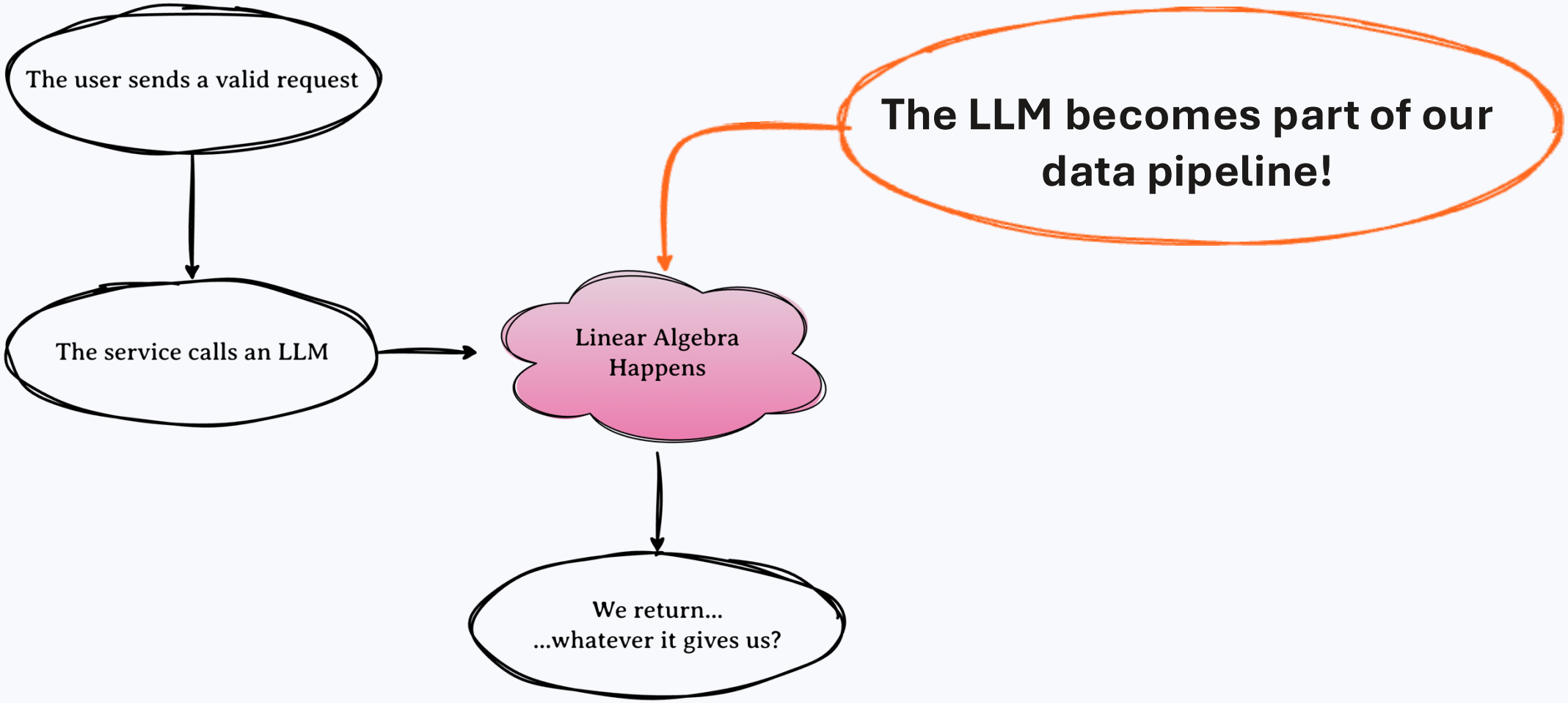
IT JUST WORKS…

…if our code produces the data.

But what happens when the data comes from an LLM?

Breaking stuff with AI

This is how we do (with AI)



This is how we do (with AI)

The moment the LLM generates the response, we lose all our guarantees

This is how we do (with AI)

- JSON that ***almost*** parses
- Missing required fields
- Extra fields we did not ask for
- Numbers represented as strings
- …

In short: **We have broken the API contract**

Can we fix it?

“We’ll just improve the prompt!”

Can we fix it?

“We’ll just improve the prompt!”

Can we fix it?

“We’ll just improve the prompt!”

- Prompts reduce the probability of failure, **but they do not eliminate it!**
- They are not contracts!
- They do not fail loudly!

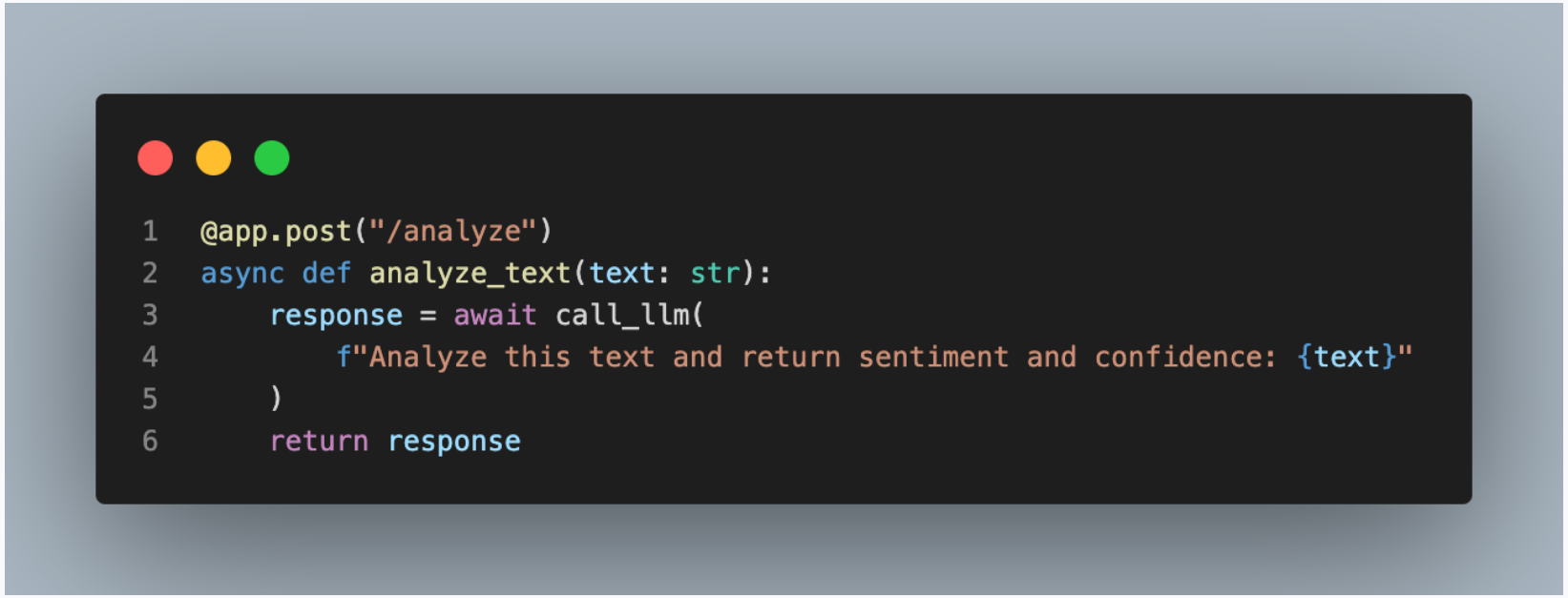
What if we treated LLM output the same way we treat user input?

What if an AI response had to earn the right to be correct and leave our API?

Enforcing Structure on LLMs

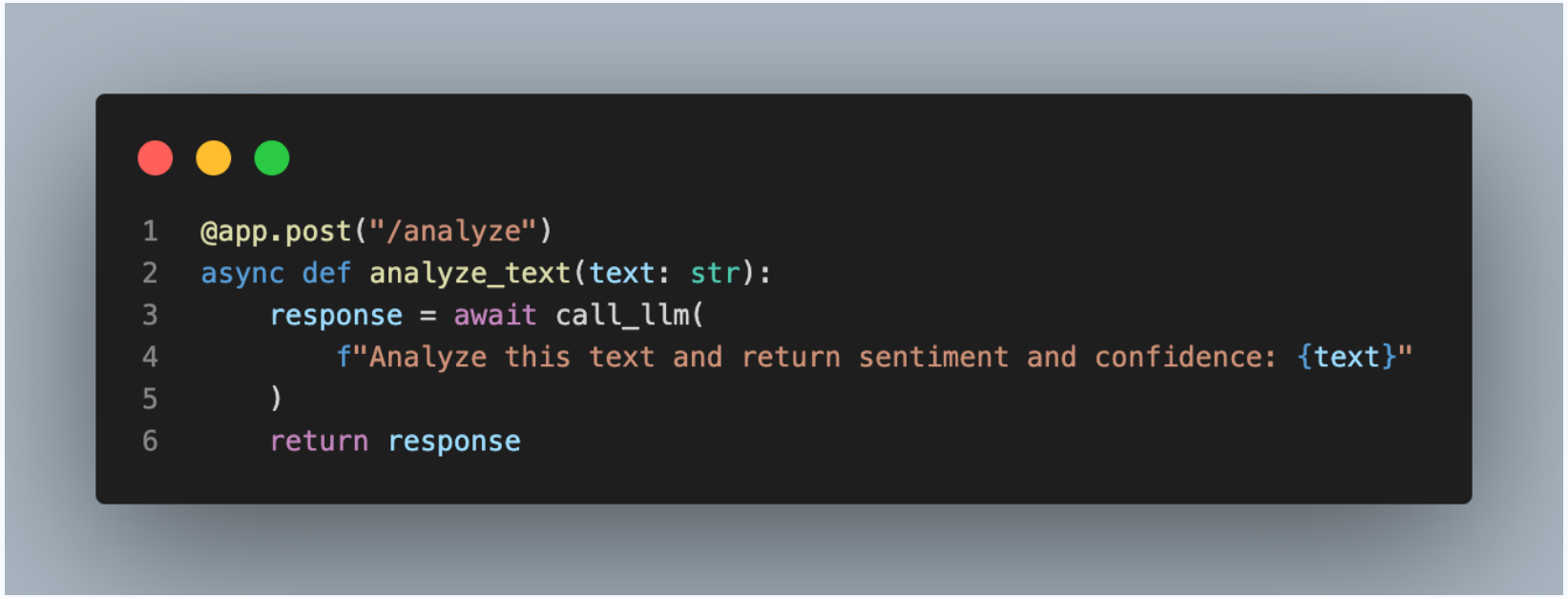
(a.k.a. *“Finally, the code!”*)

Do you wanna build an AI endpoint?



```python
@app.post("/analyze")
async def analyze_text(text: str):
    response = await call_llm(
        f"Analyze this text and return sentiment and confidence: {text}"
    )
    return response
```

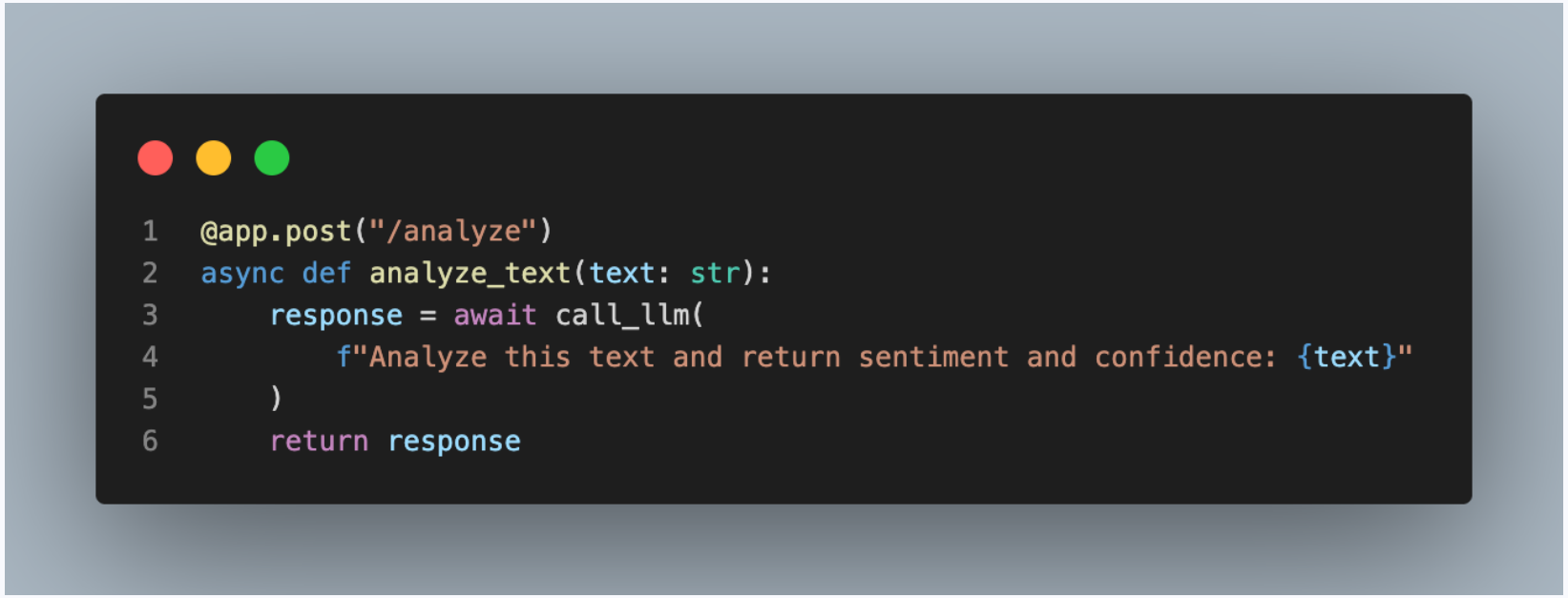
Do you wanna build an AI endpoint?



```python
@app.post("/analyze")
async def analyze_text(text: str):
    response = await call_llm(
        f"Analyze this text and return sentiment and confidence: {text}"
    )
    return response
```

What is the response type here?

Do you wanna build an AI endpoint?

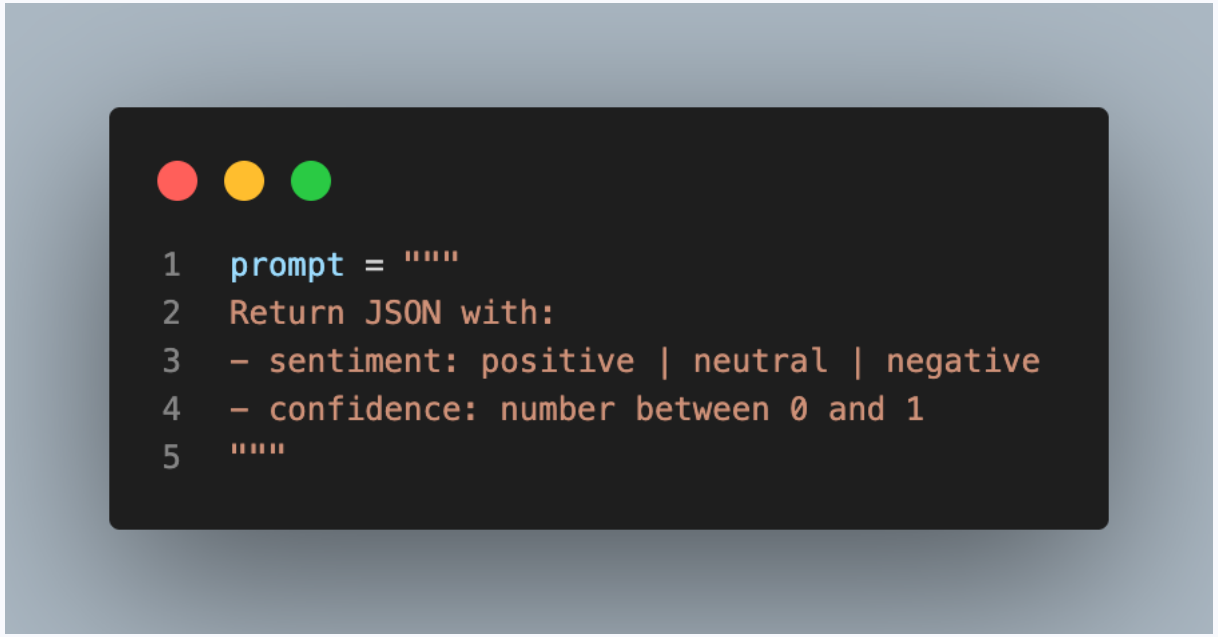


```python
@app.post("/analyze")
async def analyze_text(text: str):
    response = await call_llm(
        f"Analyze this text and return sentiment and confidence: {text}"
    )
    return response
```

What is the response type here?

We don’t know – and neither does our API!

Let’s make it a bit better



```
prompt = """
Return JSON with:
- sentiment: positive | neutral | negative
- confidence: number between 0 and 1
"""
```

This helps. A lot, actually.

TAKEAWAY

We need to treat the LLM like any untrusted external system

Let’s turn guidance into a contract

```python

from pydantic import BaseModel, Field
from typing import Literal

class SentimentResult(BaseModel):
    sentiment: Literal["positive", "neutral", "negative"]
    confidence: float = Field(ge=0.0, le=1.0)
```

First, we’ll define an output contract.

Enter PydanticAI

Pydantic AI is a Python framework for quickly and reliably building production-grade Generative AI applications and workflows

Enter PydanticAI

Features include:

1. Model-agnostic
2. Observability & Traceability out of the box
3. Full type-safety
4. Evaluation and performance monitoring
5. Support for MCP, A2A, and UI event streams
6. Human-in-the-Loop
7. Graph support for complex flows
8. And much more!

PydanticAI Agent with structured output

```python
from pydantic_ai import Agent

agent = Agent(
    model="openai:gpt-4o-mini",
    result_type=SentimentResult,
    system_prompt=(
        "Analyze the sentiment of the text and return a structured result."
    ),
)
```

THE CORE IDEA:

The LLM does not decide what is valid. The schema does.

Enter PydanticAI

```python
result = await agent.run(text)
```

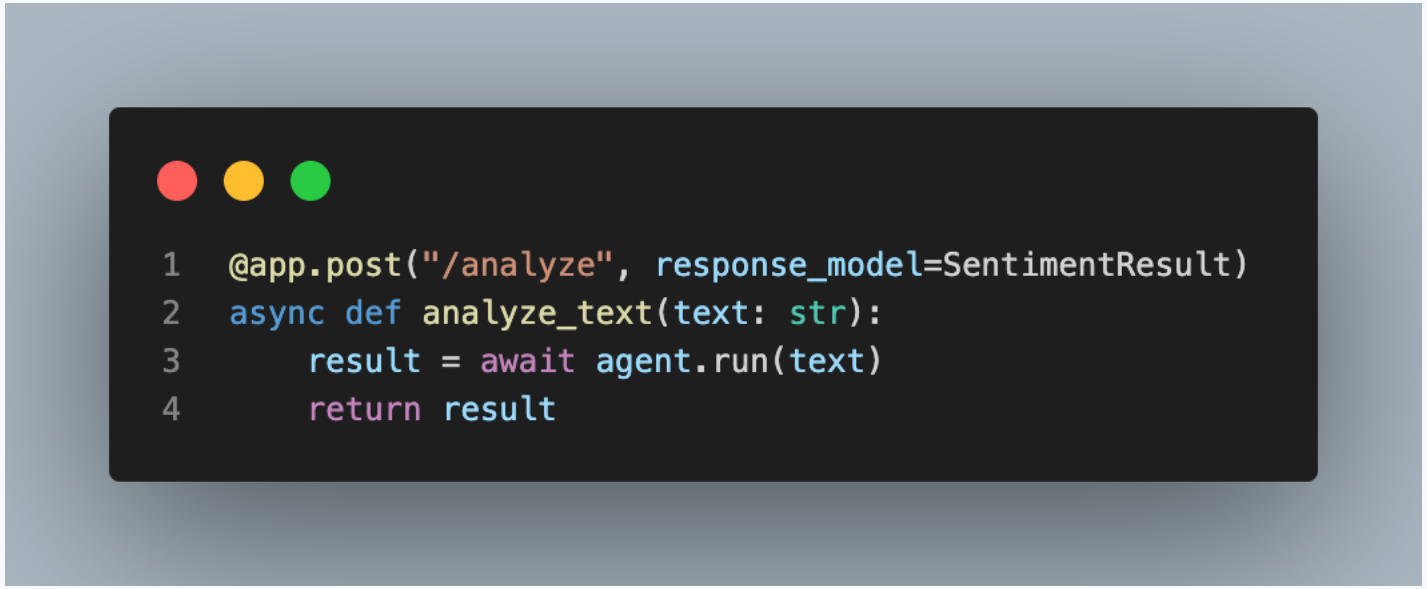
The result is not a dict. It is not a raw JSON. It is a **validated Pydantic model!**

```python
result.sentiment
result.confidence
```

BUT WHAT IF THE MODEL MISBEHAVES?

That’s fine – because failure is now visible!

Putting it all together



```python
@app.post("/analyze", response_model=SentimentResult)
async def analyze_text(text: str):
    result = await agent.run(text)
    return result
```

What we get:

- Validated request
- Validated AI output
- Validated API response
- Accurate OpenAPI schema

TAKEAWAY

The prompt helps the model. The schema protects the system.

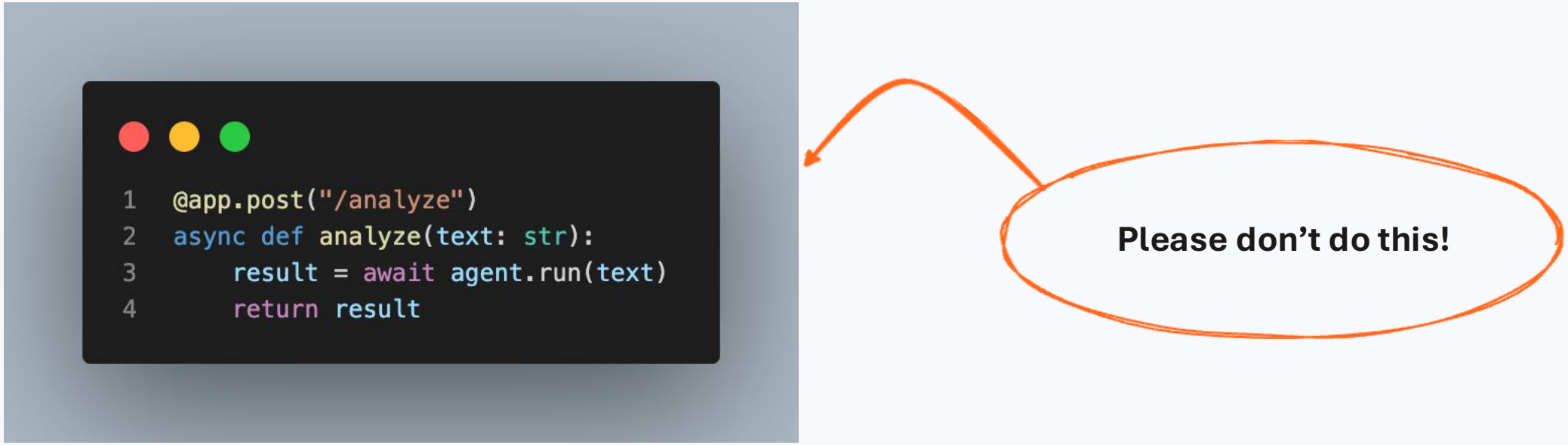
Designing usable AI APIs

Designing usable AI APIs

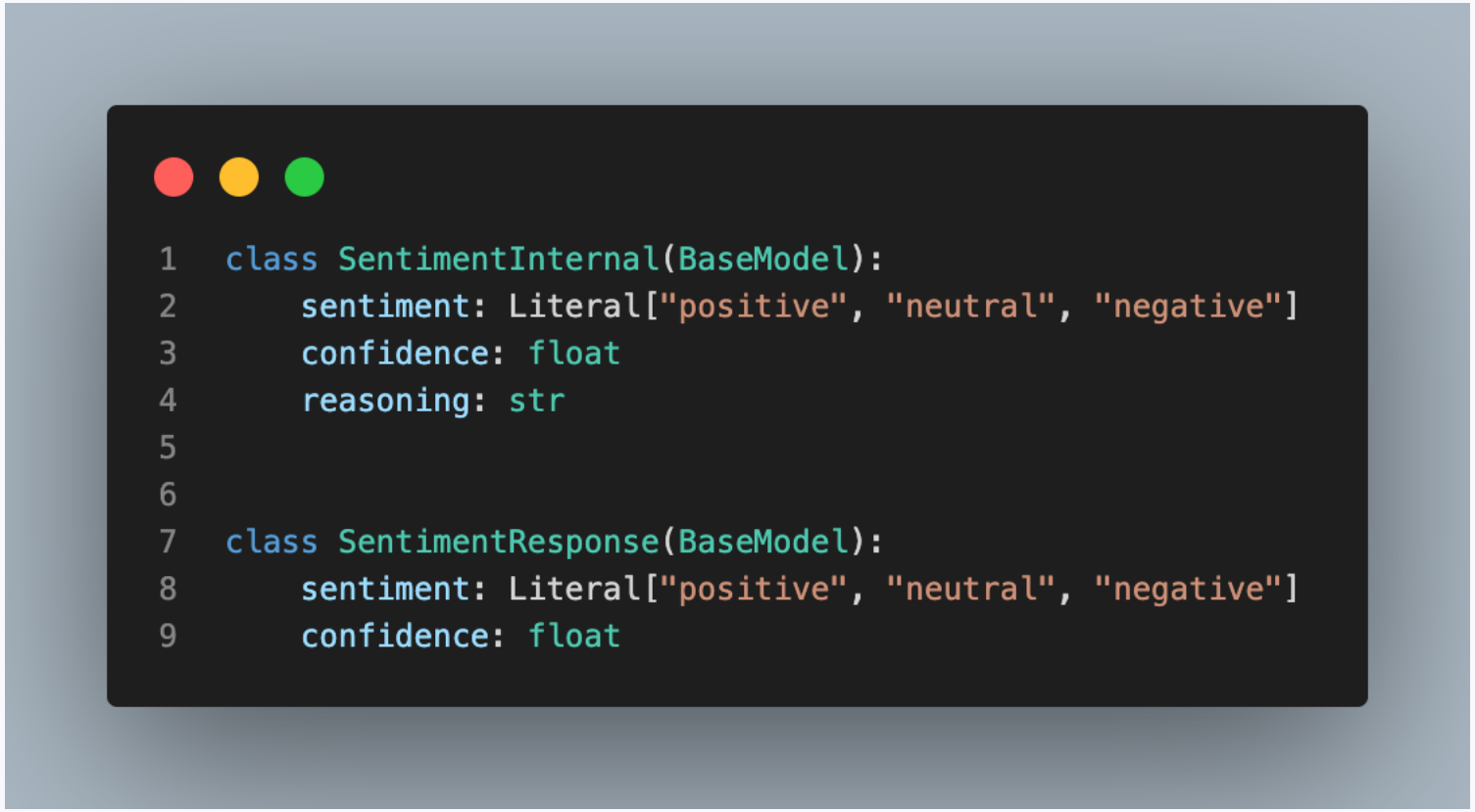
- So now our AI output is validated. Great.
- But that still doesn’t automatically make our API good.
- Validation solves correctness. Not usability. Not stability. Not long-term contracts.

An AI API still must behave like any normal API

Pattern 1: Never expose raw LLM output



Instead, separate agent output from API output



```python
class SentimentInternal(BaseModel):
    sentiment: Literal["positive", "neutral", "negative"]
    confidence: float
    reasoning: str


class SentimentResponse(BaseModel):
    sentiment: Literal["positive", "neutral", "negative"]
    confidence: float
```

Instead, separate agent output from API output – then transform!

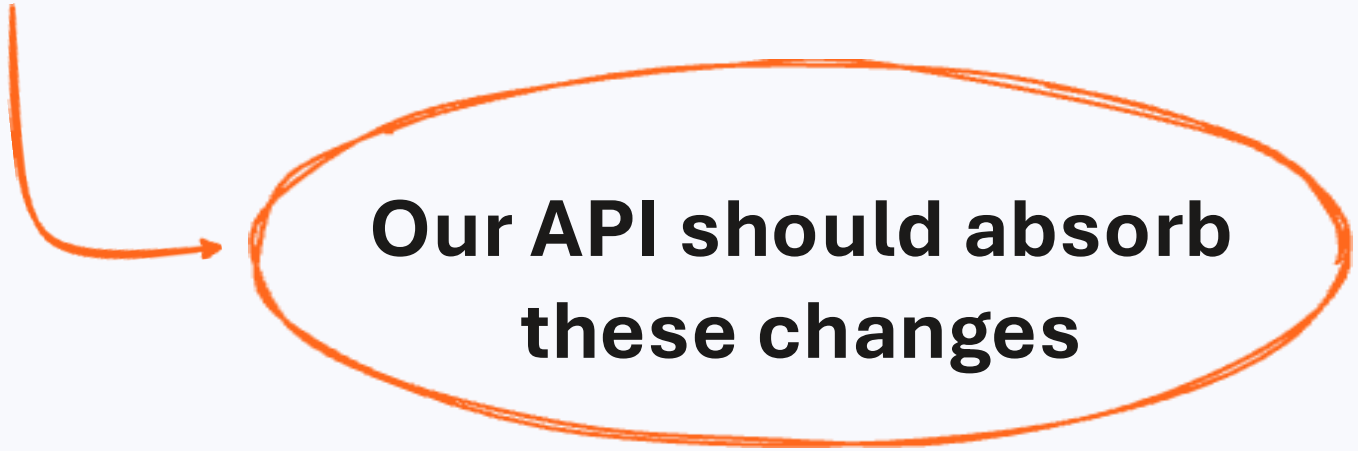
```
result = await agent.run(text)

return SentimentResponse(
    sentiment=result.sentiment,
    confidence=result.confidence
)
```

Pattern 2: Make OpenAPI honest

Your OpenAPI schema now represents exactly what clients receive

Pattern 3: Design for evolution

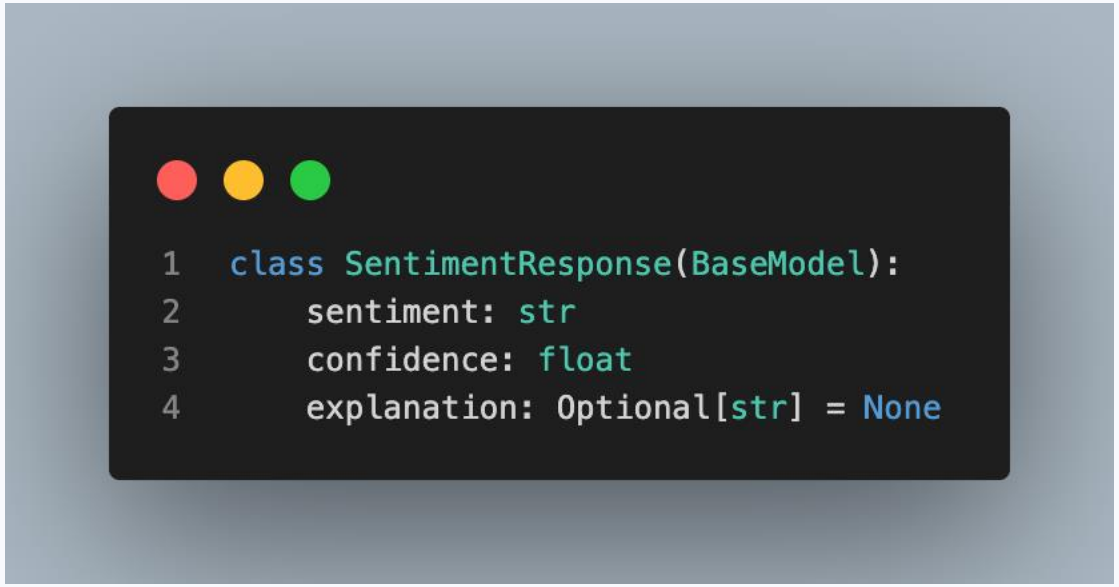
- Prompts change
- Models change
- Tools change



Pattern 3: Design for evolution

Techniques:

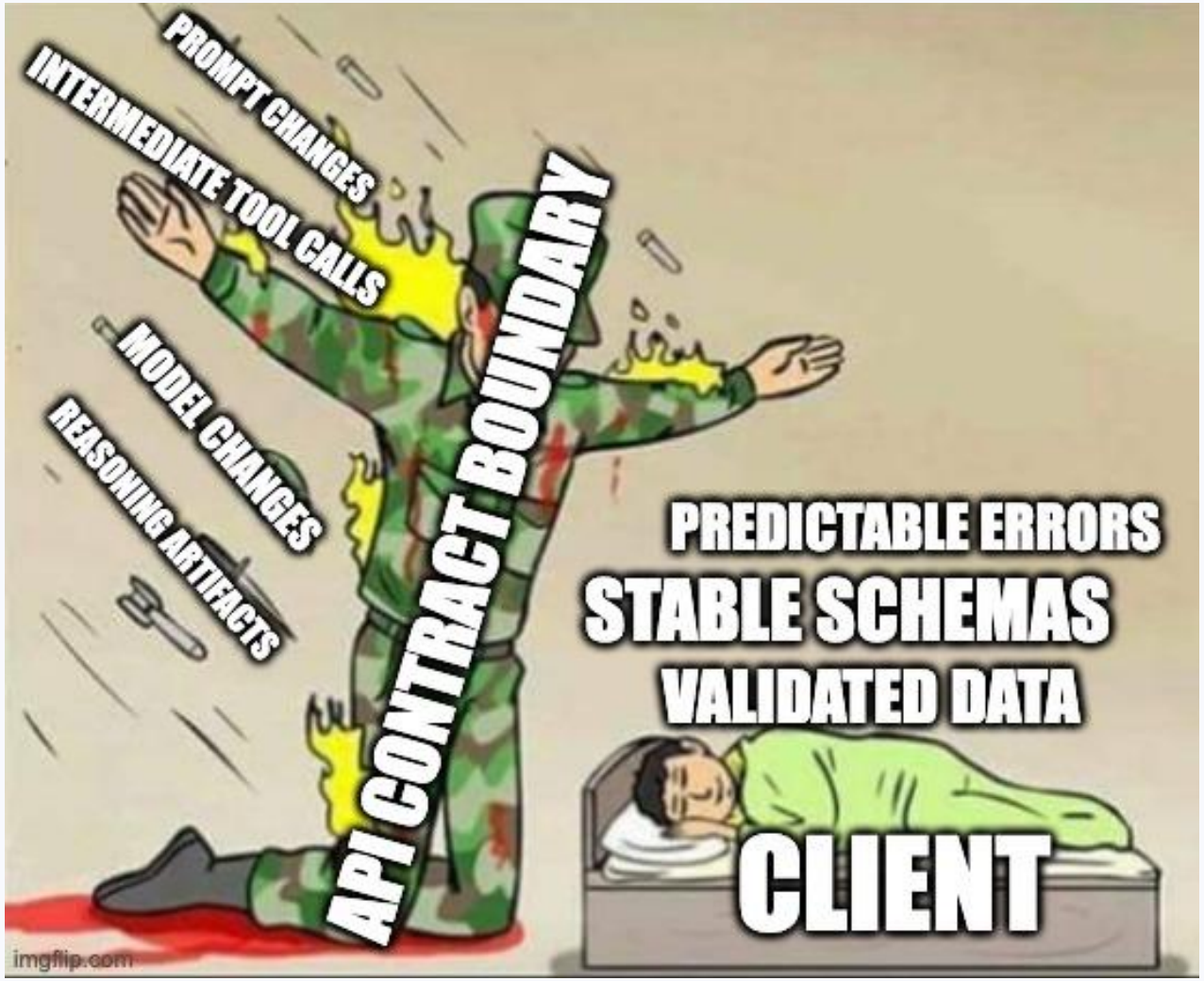
- Stable response models
- Optional fields for expansion
- Versioned endpoints when needed



```python
class SentimentResponse(BaseModel):
    sentiment: str
    confidence: float
    explanation: Optional[str] = None
```

The AI can get smarter without breaking clients

The bigger design principle



The bigger design principle

Once schemas define :
everything

- Validation
- Documentation
- AI output
- API responses

We get a powerful feedback loop!

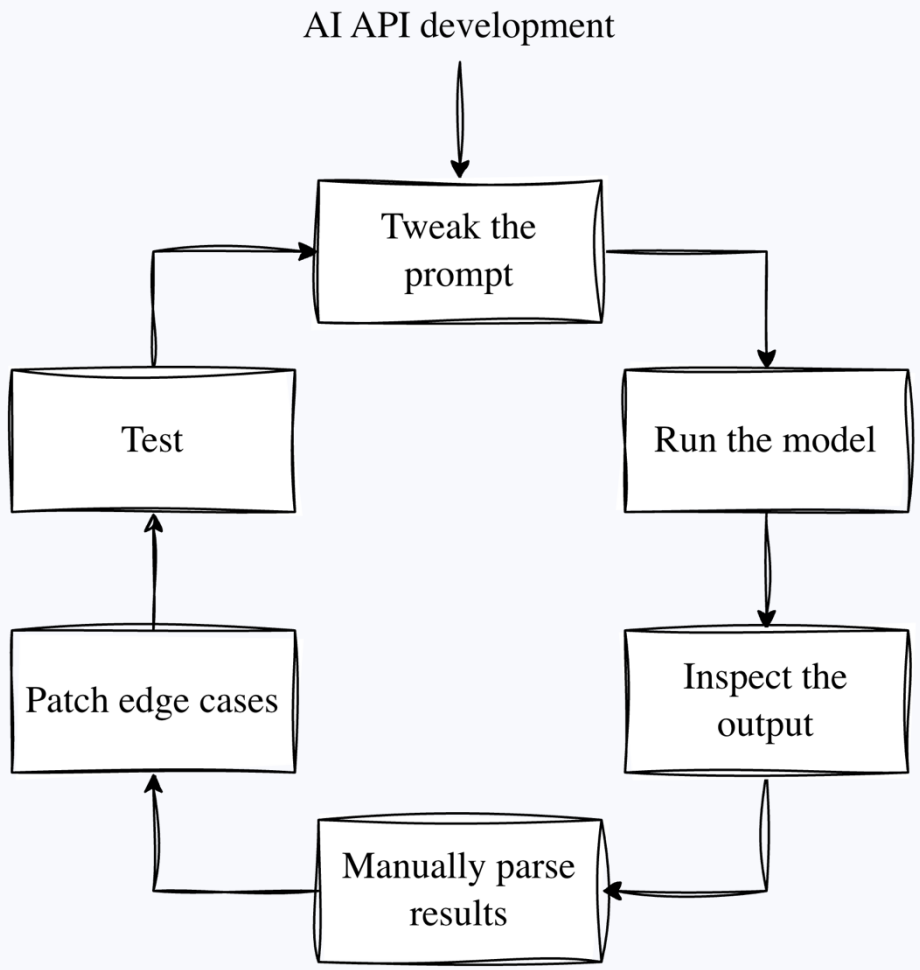
This is where the Python ecosystem becomes a developer’s best friend

AND THE SURPRISING THING IS:

This does not slow developers down; **it makes them faster!**

How this affects DevEx

AI development normally feels chaotic

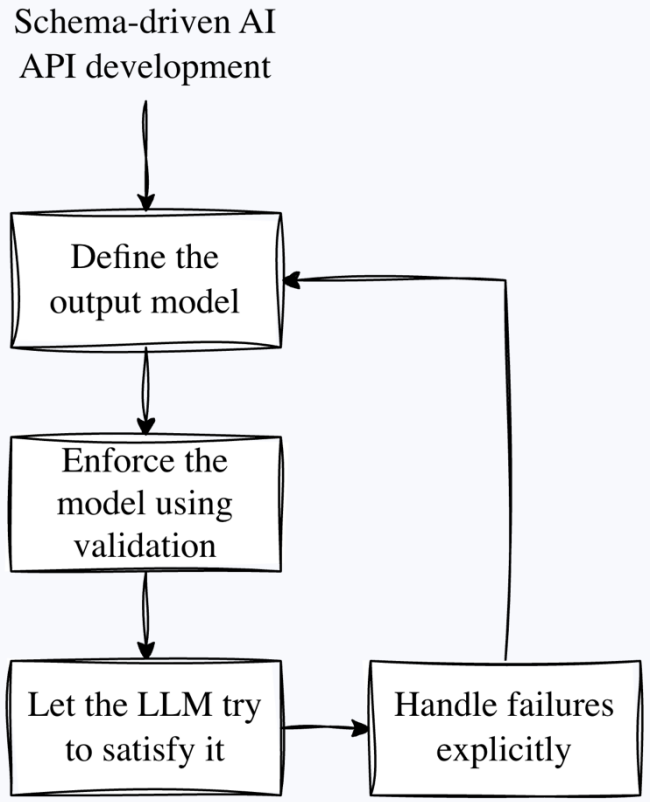


This leads to:

- Fragile parsing logic
- Implicit assumptions
- Constant regressions
- Hard-to-debug failures

... it does not feel like software engineering at all!

Let’s reframe using what we’ve learned



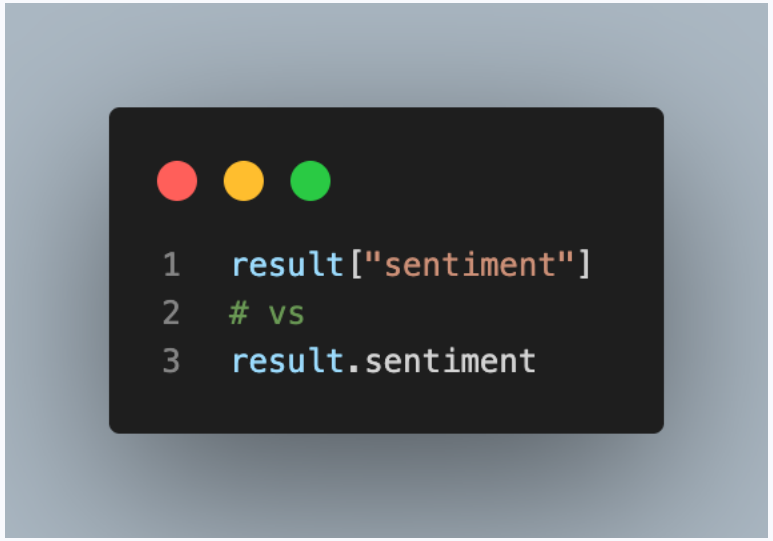
Now our development loop is:

- Deterministic
- Testable
- Inspectable

… instead of chasing weird outputs, you define what “correct” means!

The hidden productivity benefits

- **Autocomplete**
- **Safer refactors**
  - The IDE becomes our ally
- **Faster collaboration**
  - Schema as a shared language
- **Reliable documentation**
  - OpenAPI is always accurate – no doc drift!



THE PSYCHOLOGICAL SHIFT

“How do I make the **model behave**?”

“What does a **correct answer** look like?”

With the right tools, it's easy

In many ecosystems, achieving this requires a lot of glue code.

In Python, it mostly falls out of the tools

TAKEAWAY

Schemas do not slow down AI development – **they make it safe to move fast!**

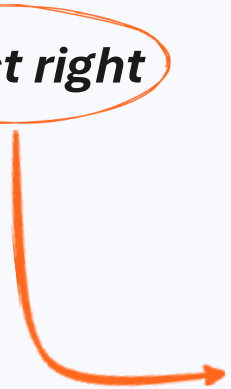
Make AI APIs boring again

Let’s recap

- Traditional APIs are ***deterministic***
- AI systems are ***probabilistic***
  - LLMs sometimes produce output that is ***almost right***

Let’s recap

- Traditional APIs are ***deterministic***
- AI systems are ***probabilistic***
  - LLMs sometimes produce output that is ***almost right***



This is where most problems begin

Let’s recap – the core mistake

- When building AI APIs, we often focus on the model
  - *Better prompts*
  - *Better context*
  - *Better tools*

Let’s recap – the core mistake

- When building AI APIs, we often focus on the model
  - *Better prompts*
  - *Better context*
  - *Better tools*

But reliability does not come from prompts – **reliability comes from contracts**

Let’s recap – the grand idea

The AI can be creative internally – the API should never be!

So, please:

- Validate your inputs
- Enforce schemas on outputs
- Separate internal models from public APIs
- Treat LLMs as untrusted dependencies

FINAL TAKEAWAY

Prompt engineering helps the **model understand the task**.

Schema engineering helps the **system stay trustworthy!**

In practice, schema beats prompt – every time.

Thank you!

Edvin Teskeredzic

Senior AI Software Engineer

Edvin.Teskeredzic@infobip.com

www.infobip.com

LET'S STAY IN TOUCH