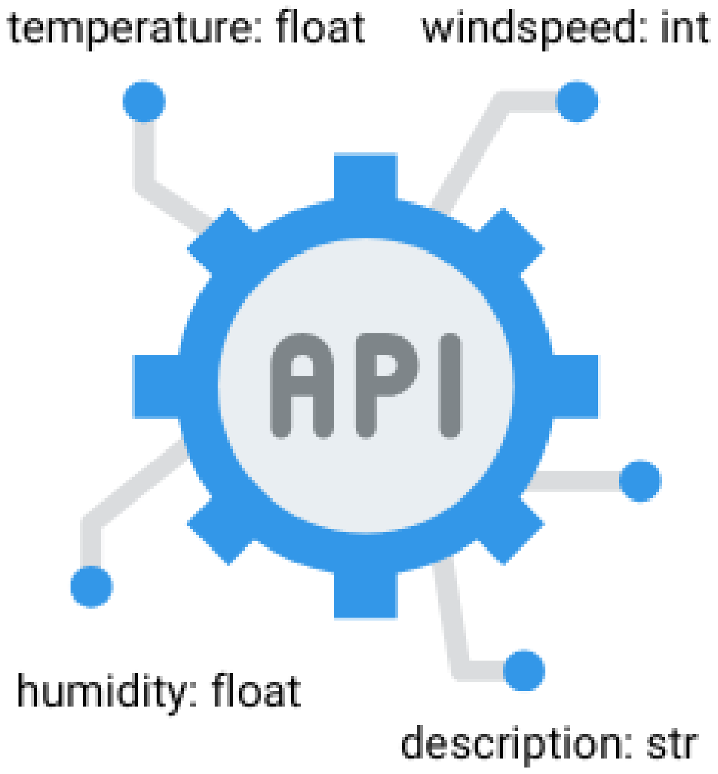
# Handling different input types in FastAPI

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

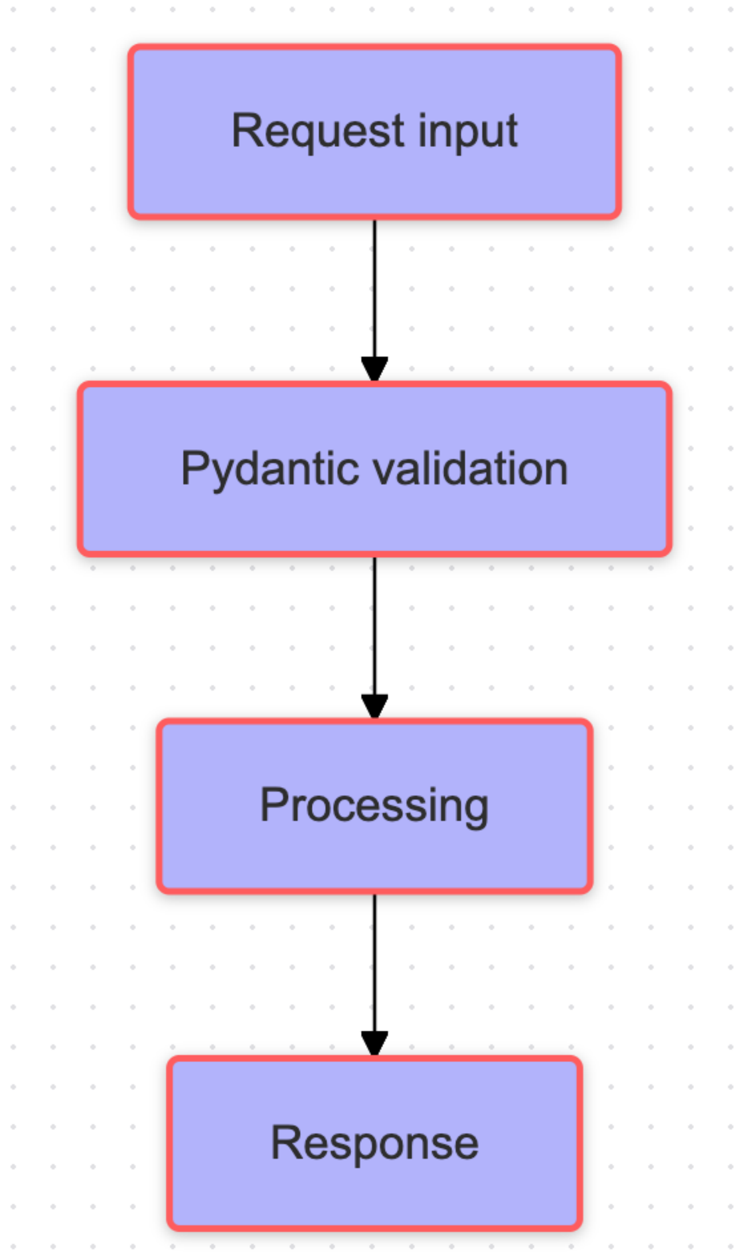
**Matt Eckerle**
Software and Data Engineering Leader

# Restaurant vs API



# Validation flow

- Incoming data via request
- Input data validation happens using Pydantic
- Process different types of data as per model requirements
- Processed input sent to the model



# Comment moderation system

```
class CommentMetrics(BaseModel):
    length: int
    user_karma: int
    report_count: int
class CommentText(BaseModel):
    content: str
```

# Endpoint for floating point numbers

```
app = FastAPI()
@app.post("/predict")
def predict_score(data: CommentMetrics):
    features = np.array([
        data.length,
        data.user_karma,
        data.report_count
    ])
    model = CommentScorer()
    prediction = model.predict(features)
    return {"prediction": round(prediction, 2),
            "input": data.dict()}
```

# Endpoint for textual input

```python
@app.post("/analyze_text")
def analyze(comment: CommentText):
    forbidden = ["spam", "hate", "free"
                 "fake", "sign up"]
    text_lower = comment.lower()
    issues = [word for word in forbidden
              if word in text_lower]
    return {
        "issues": issues,
        "needs_moderation": len(issues)
    }
```

Output for comment: `Sign up for free`

```
{
    "issues": ["free", "sign up"],
    "needs_moderation": 2
}
```

# Let's practice!

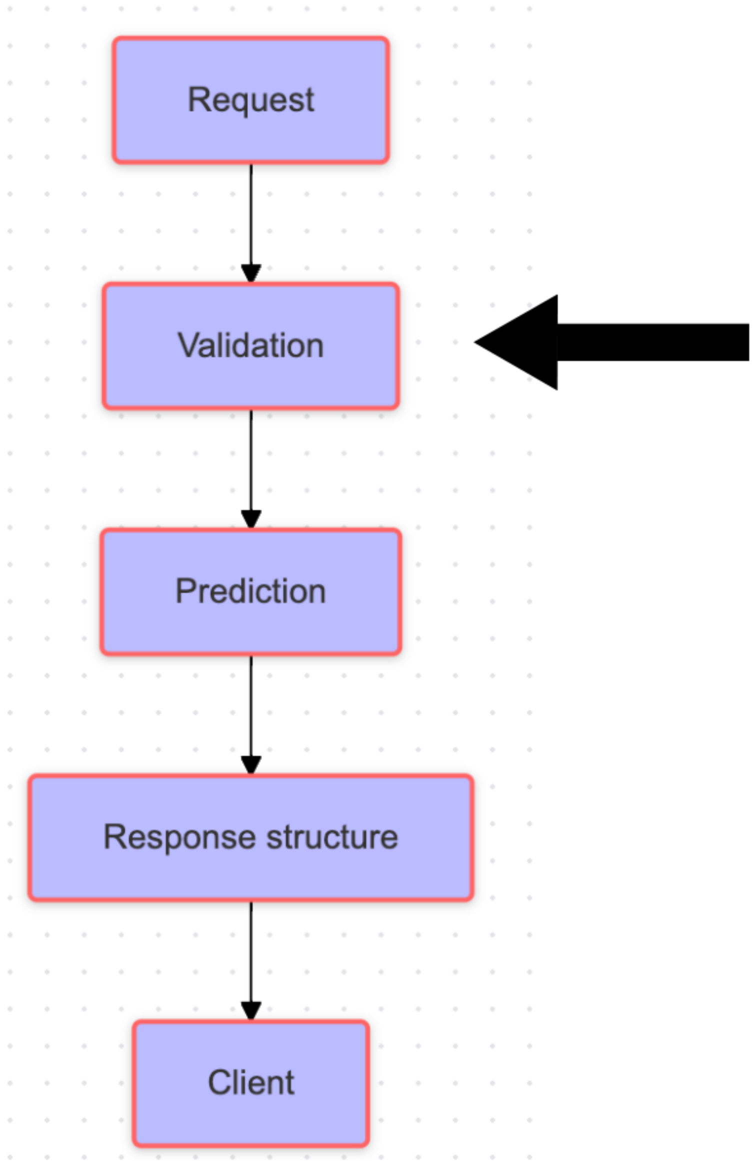
DEPLOYING AI INTO PRODUCTION WITH FASTAPI

# Input validation in FastAPI

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

**Matt Eckerle**
Software and Data Engineering Leader

# Validating input data



# Why validate the input?

- Validation for data integrity
- Prevent errors in the application
- Integrates with Pydantic
- Provided powerful tools for data validation

# Pydantic for pre-defined function

**Field Validators**

The `Field` function is used to customize and add metadata to fields of models

# Custom validation with pydantic

⚡

### Field Validators

The `Field` function is used to customize and add metadata to fields of models

### Custom Domain-Specific Validators

Create and apply custom validator functions

# Graceful error reporting

⚡

**Field Validators**

The `Field` function is used to customize and add metadata to fields of models

🛡️

**Custom Domain-Specific Validators**

Create and apply custom validator functions

⚠️

**Validation Error Handling**

Custom messages and user-friendly reporting

# Pydantic field validators

- User registration endpoint
- Validating the username entered by users:

```
from pydantic import BaseModel, Field
```

```
class User(BaseModel):
    username: str = Field(..., min_length=3, max_length=50)
```

# Adding custom validators

```
class User(BaseModel):
    username: str = Field(...,
                          min_length=3,
                          max_length=50)
    age: int
    @field_validator('age')
    def age_criteria(cls, age):
        if age < 13:
            raise ValueError('User must be at least 13')
        return age
```

# Custom validators in action

## Valid request:

```
{"username": "john_doe", "age": 25}
```

```
Valid user: username='john_doe' age=25
```

## Invalid request:

```
{"username": "too_young", "age": 10}
```

```
Validation error for {'username': 'too_young', 'age': 10}: User must be at least 13
```

# Putting it all together

**Field Validators**

The `Field` function is used to customize and add metadata to fields of models

**Custom Domain-Specific Validators**

Create and apply custom validator functions

**Validation Error Handling**

Custom messages and user-friendly reporting

- Field validator for username
- Custom validator for age
- Error message if failing validation

# Putting it all together

```python
@app.post("/users")
def create_user(user: User):
    return {"message": "User created",
            "user": user.model_dump()}
```

**Output:**

```
{
  "message": "User created successfully",
    "user": {
          "username": "john_doe",
           "age": 25
    }
}
```

# Let's practice!

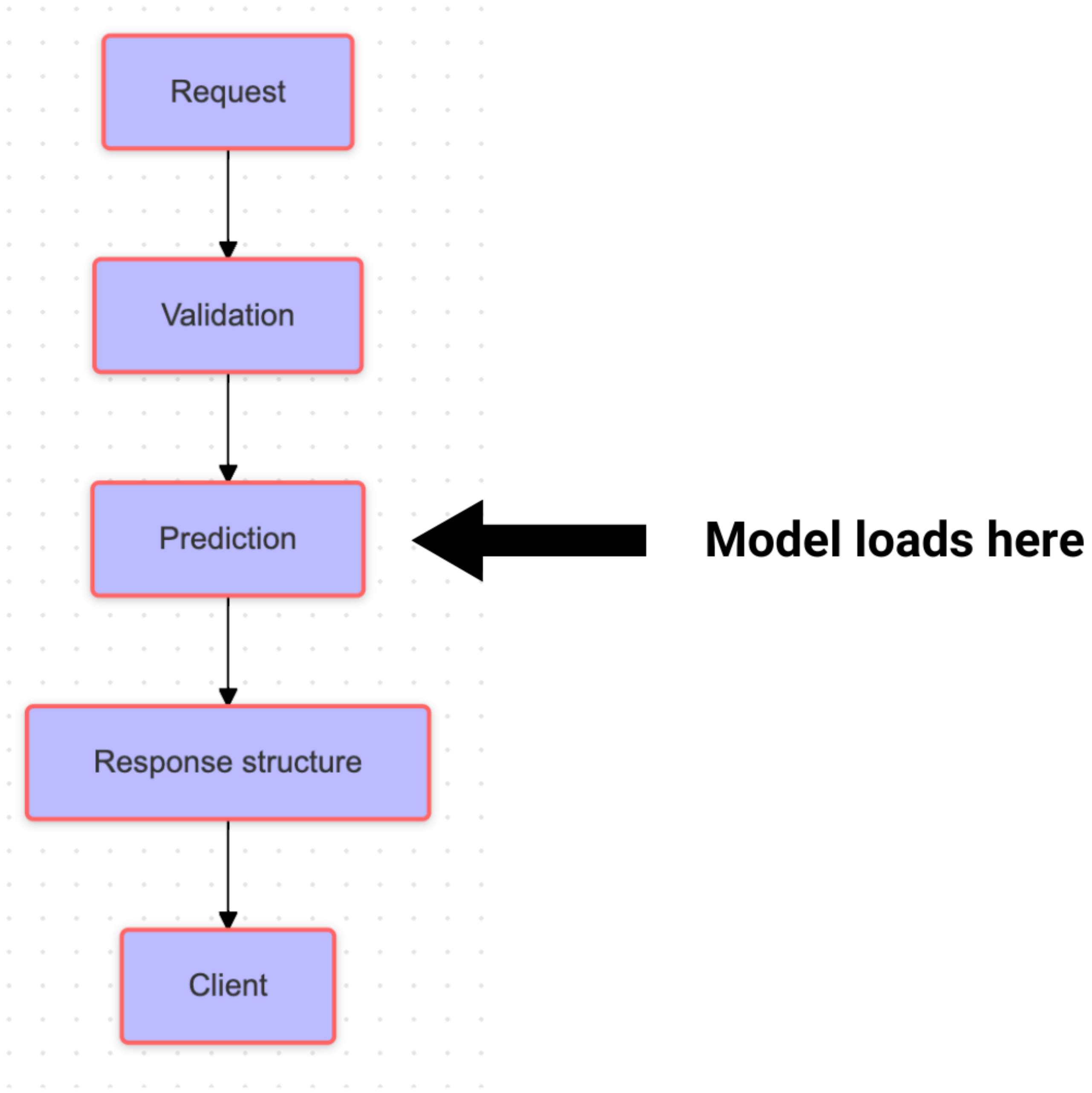
DEPLOYING AI INTO PRODUCTION WITH FASTAPI

# Loading a pre-trained model

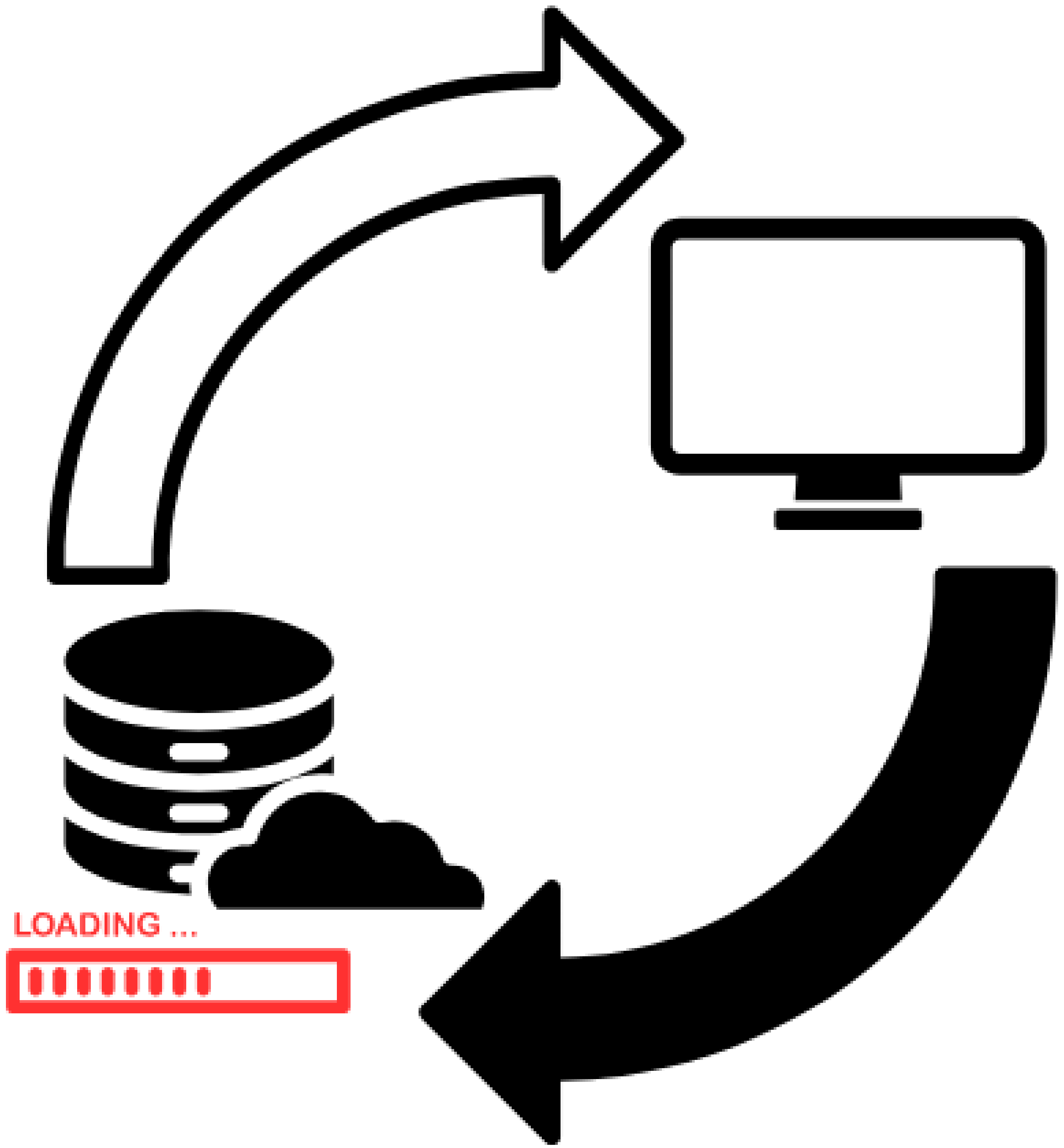
DEPLOYING AI INTO PRODUCTION WITH FASTAPI

**Matt Eckerle**
Software and Data Engineering Leader

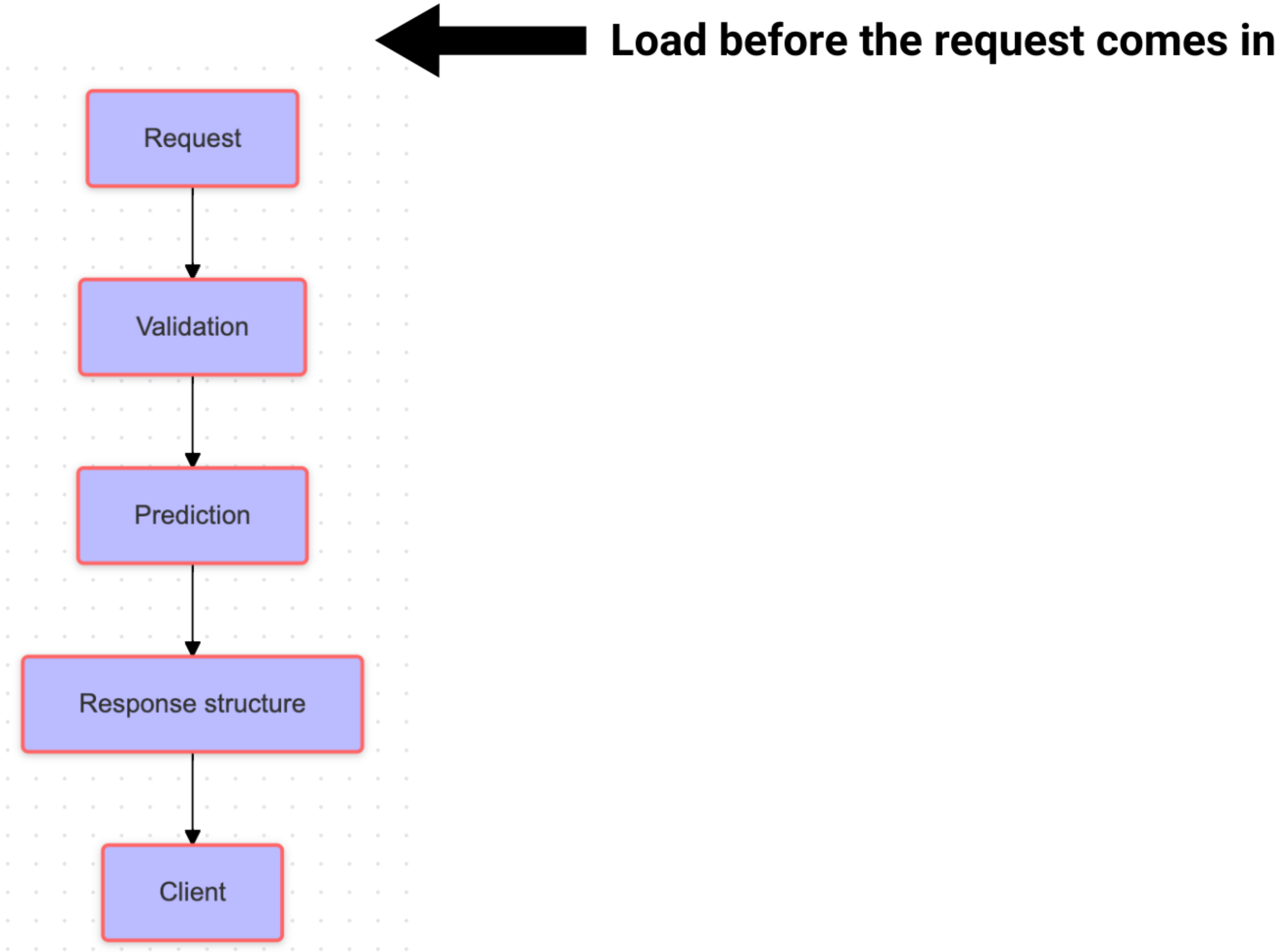
# Current structure



# Challenge with loading models




# Load models before the request



# Loading the model

```python
from fastapi import FastAPI
```

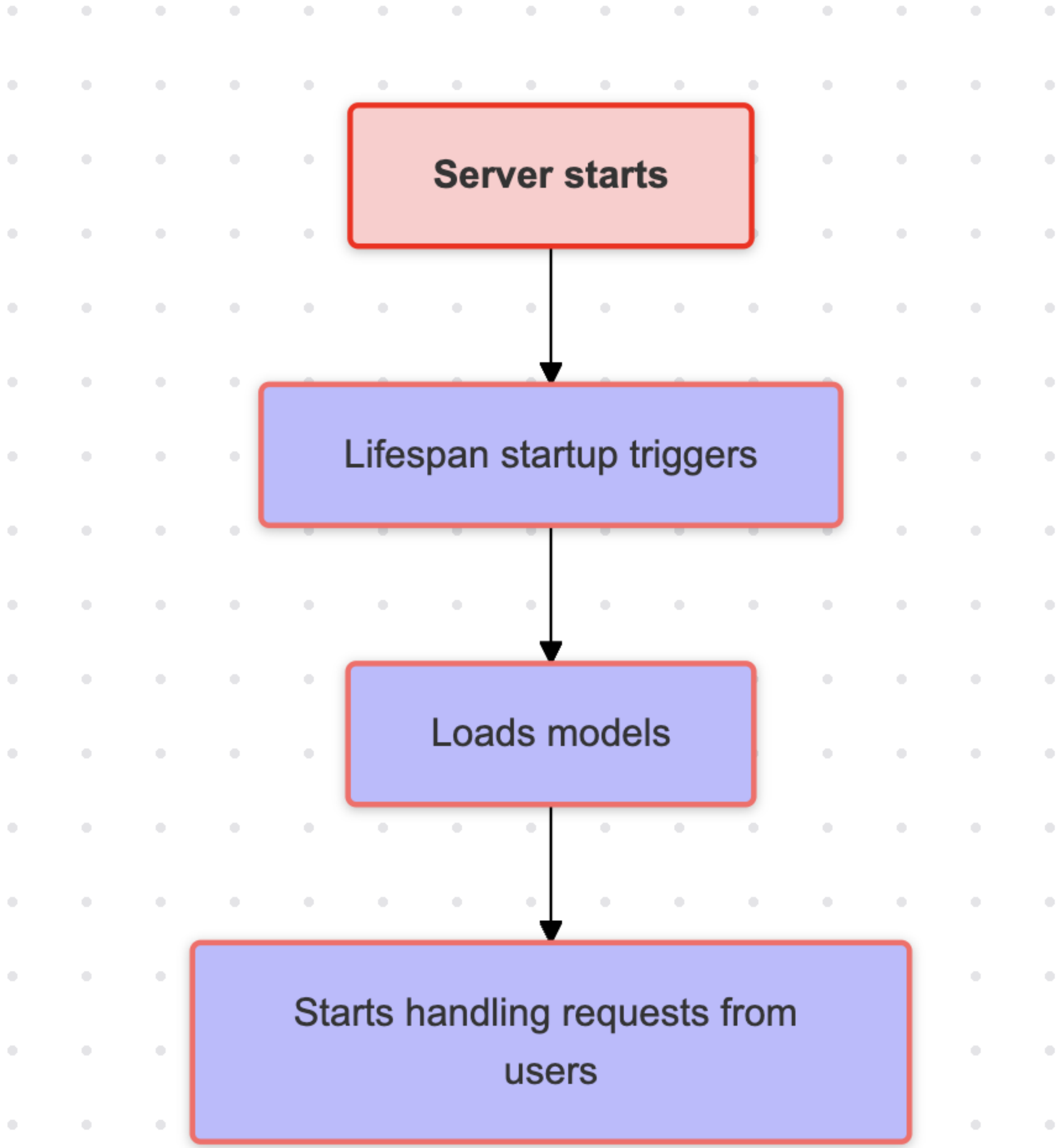
```python
sentiment_model = None
```

```python
def load_model():
    global sentiment_model
    sentiment_model = SentimentAnalyzer("trained_model.joblib")
    print("Model loaded successfully")
```

```python
load_model()
```

```
Model loaded successfully
```

# FastAPI lifespan event



# FastAPI lifespan event

```python
from contextlib import asynccontextmanager
```

```python
@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup: Load the ML model
    load_model()
    yield
```

```python
app = FastAPI(lifespan=lifespan)
```

# Health checks

```python
@app.get("/health")
def health_check():
    if sentiment_model is not None:
        return {"status": "healthy",
                "model_loaded": True}
    return {"status": "unhealthy",
            "model_loaded": False}
```

Curl command:

```
curl -X GET \
  "http://localhost:8080/health" \
  -H "accept: application/json"
```

**Output:**

```
{
  "status": "healthy",
  "model_loaded": true
}
```

# Let's practice!

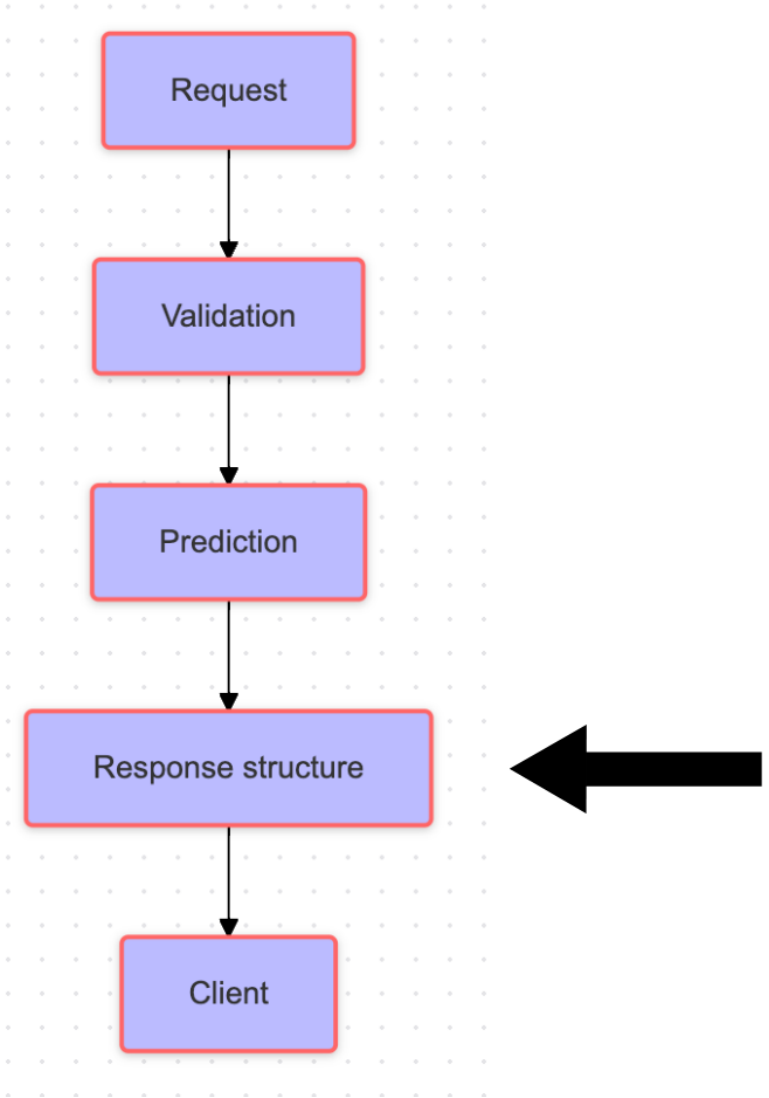
DEPLOYING AI INTO PRODUCTION WITH FASTAPI

# Returning structured prediction response

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Matt Eckerle
Software and Data Engineering Leader

# Challenges with deploying models




1. Accept input data properly
2. Validate incoming data and handle errors
3. Make predictions
4. Return well-structured responses

# Defining request structure

```python
from pydantic import BaseModel
```

```python
class PredictionRequest(BaseModel):
    text: str
```

```python
class PredictionResponse(BaseModel):
    text: str
    sentiment: str
    confidence: float
```

# Creating the prediction endpoint

```python
@app.post("/predict")
def predict_sentiment(request: PredictionRequest):
    if sentiment_model is None:
        raise HTTPException(
            status_code=503,
            detail="Model not loaded"
        )
    result = sentiment_model(request.text)
    return PredictionResponse(
            text=request.text,
            sentiment=result[0]["label"],
            confidence=result[0]["score"]
        )
```

Input JSON:

```json
{"text": "This movie was fantastic!"}
```

Response:

```json
{
    "text": "This movie was fantastic!",
    "sentiment": "POSITIVE",
    "confidence": 0.95
}
```

# Error handling

```
try:
  result = sentiment_model(request.text)
  return PredictionResponse(
        text=request.text,
        sentiment=result[0]["label"],
        confidence=result[0]["score"]
    )
except Exception:
  raise HTTPException(
        status_code=500,
        detail="Prediction failed"
    )
```

Response when model fails to predict

```
{
    "detail": "Prediction failed",
    "status_code": 500
}
```

# Testing the endpoint

```python
# Example request
import requests

response = requests.post(
    "http://localhost:8000/predict",
    json={"text": "Great product!"}
)
print(response.json())
```

```
{
    "text": "Great product!",
    "sentiment": "POSITIVE",
    "confidence": 0.998
}
```

# Let's practice!

DEPLOYING AI INTO PRODUCTION WITH FASTAPI