API Key Authentication

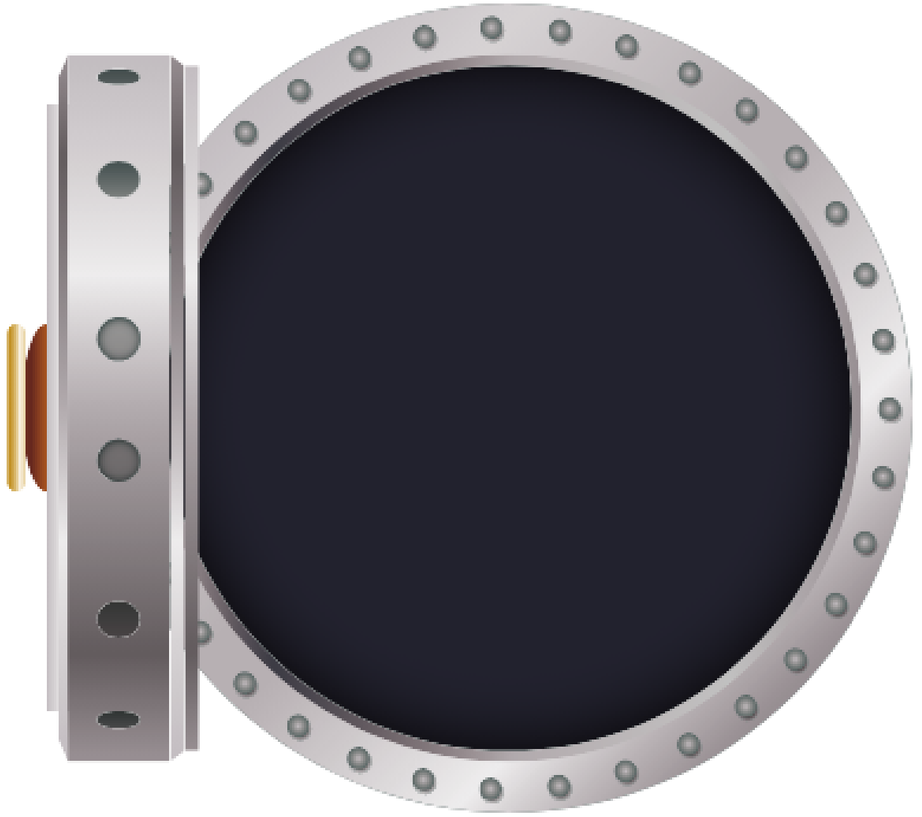
DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Matt Eckerle
Software and Data Engineering Leader

datacamp

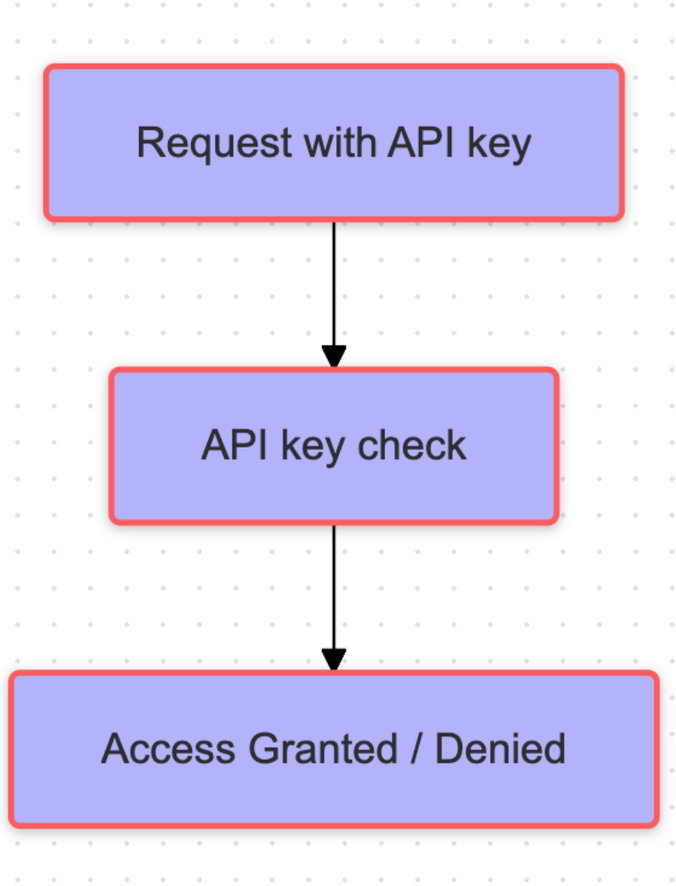
Why secure APIs?

- Stop unauthorized users
- Secure API endpoints with API key authentication



How API keys work

- Like a digital password for our API
- Sent in request headers
- Verified before accessing endpoints



Understanding APIKeyHeader

```
from fastapi import FastAPI
from fastapi.security import APIKeyHeader
header_scheme = APIKeyHeader(
                        name="X-API-Key",
                        auto_error=True
                    )
```

Authenticating an endpoint

```python
from fastapi.security import APIKeyHeader
from fastapi import Depends, HTTPException

header_scheme = APIKeyHeader(name="X-API-Key",
                             auto_error=True)

API_SECRET_KEY = "your-secret-key"

@app.get("/items/")
def read_items(
    api_key: str = Depends(header_scheme)
):
    if api_key != API_SECRET_KEY:
        raise HTTPException(
            status_code=403,
            detail="Invalid API key")
    return {"api_key": api_key}
```

- `ApiKeyHeader`
- `Depends` adds header scheme
- `HTTPException` for exceptions
- Defines API key header and secret key
- Validates API keys with `test_api_key`
- Raises `403` if the key doesn't match `API_SECRET_KEY`

Authenticating an app

```
def verify_api_key(api_key: str = Depends(header_scheme)):
    if api_key != API_KEY:
        raise HTTPException(status_code=403,  detail="Invalid API key")
    return api_key

app = FastAPI(
    dependencies=[Depends(verify_api_key)]
)
@app.post("/predict")
def predict_sentiment(text: str):
    return {
        "text": text,
        "sentiment": "positive",
        "status": "success"
    }
```

Testing the endpoint

Command with invalid API key:

```
curl -X POST \
  http://localhost:8000/predict \
  -H "X-API-Key: wrong-key" \
  -H "Content-Type: application/json" \
  -d '{"text": "This product is amazing!"}'
```

Invalid key output:

```
{"detail":"Invalid API key"}
```

Command with valid API key:

```
curl -X POST \
  http://localhost:8000/predict \
  -H "X-API-Key: your-secret-key" \
  -H "Content-Type: application/json" \
  -d '{"text": "This product is amazing!"}'
```

Valid key output:

```
{"text":"This product is amazing!",
 "sentiment":"positive",
 "status":"success"}
```

Let's practice!

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

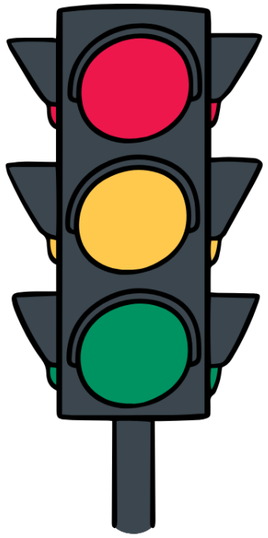
Rate Limiting

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Matt Eckerle
Software and Data Engineering Leader

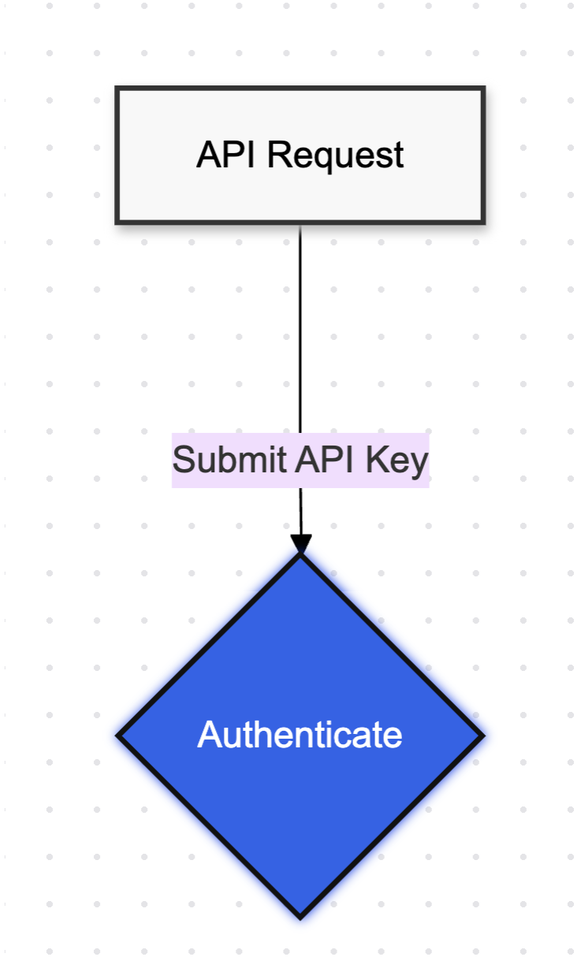
datacamp

Introducing rate limiting

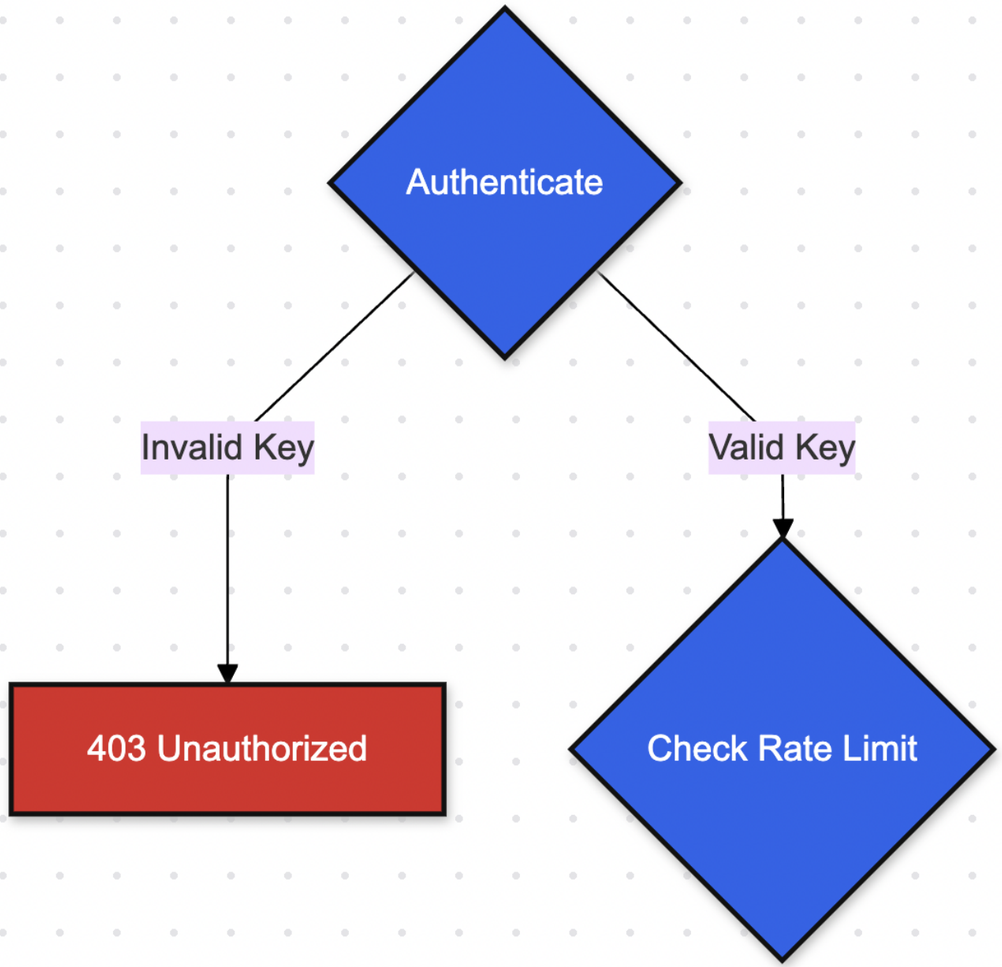


- **Purpose:** Controls the frequency of API requests.
- **Response:** Returns HTTP 429 ("Too Many Requests") when the limit is exceeded.

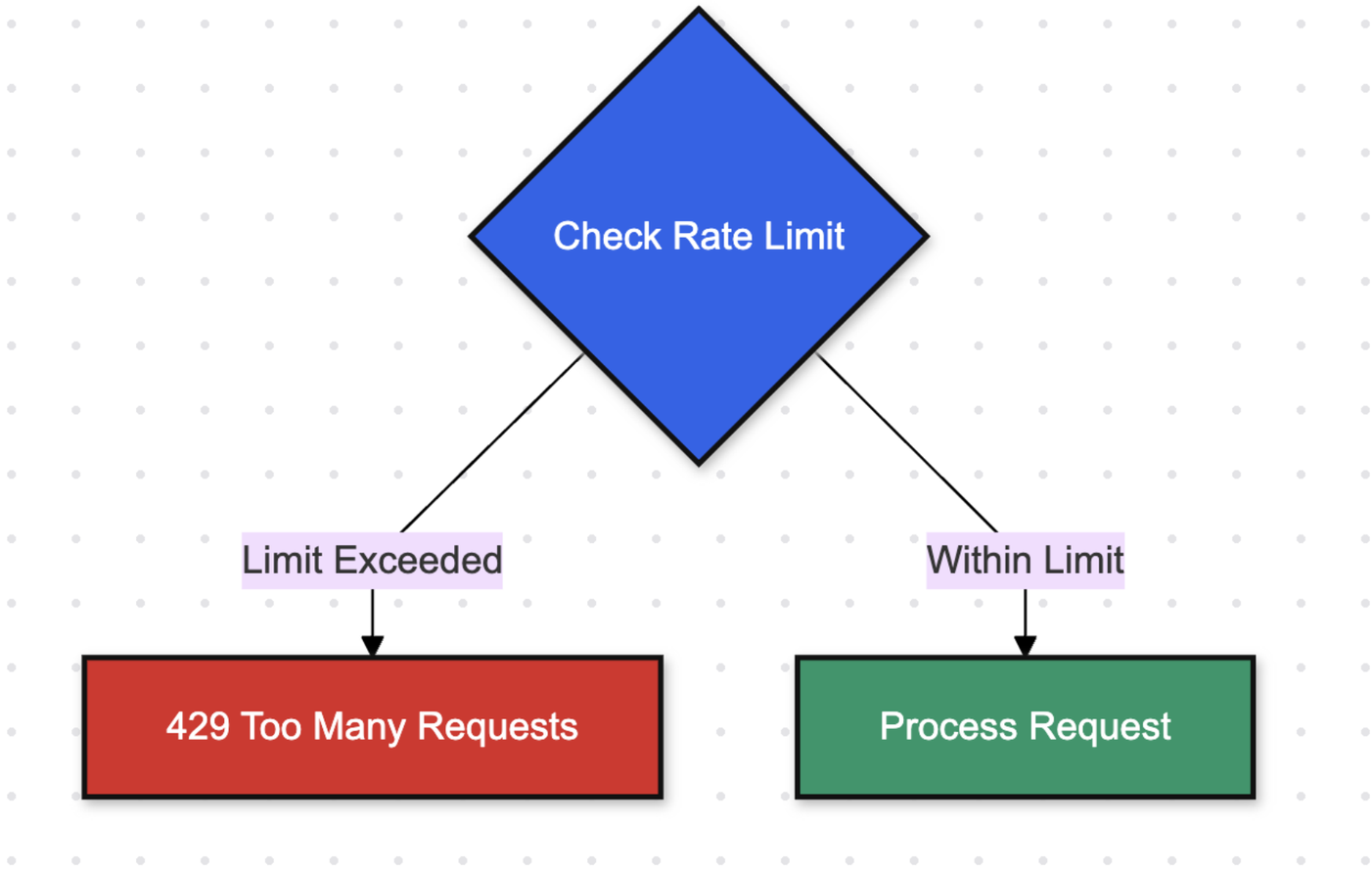
How rate limiting works



Authenticating incoming credentials



Rate limiting check

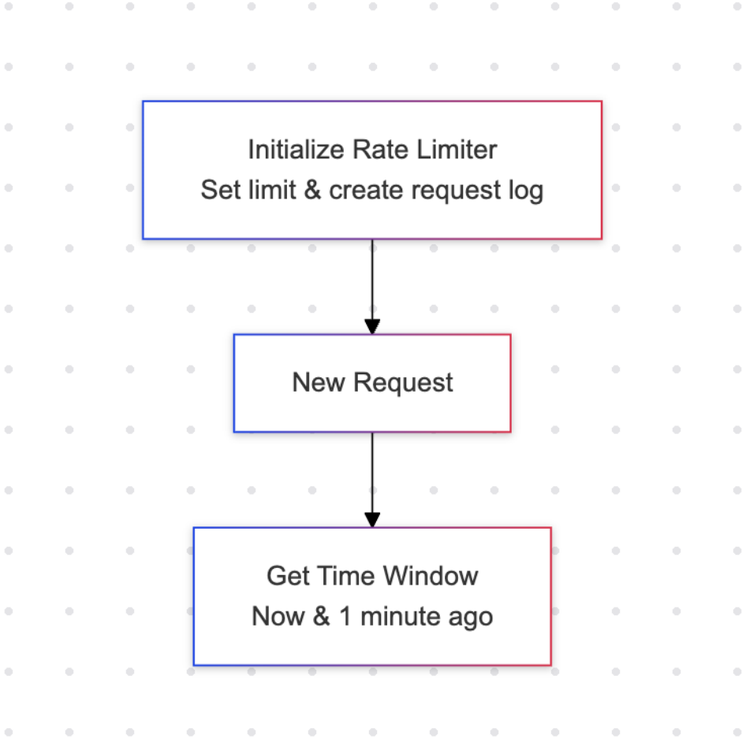


Setting up our API

```
from fastapi import FastAPI, Depends, HTTPException
from fastapi.security import APIKeyHeader
from pydantic import BaseModel

app = FastAPI()
model = SentimentAnalyzer(pkl_file_path)

API_KEY_HEADER = APIKeyHeader(name="X-API-Key")
API_KEY = "your-secret-key"
```

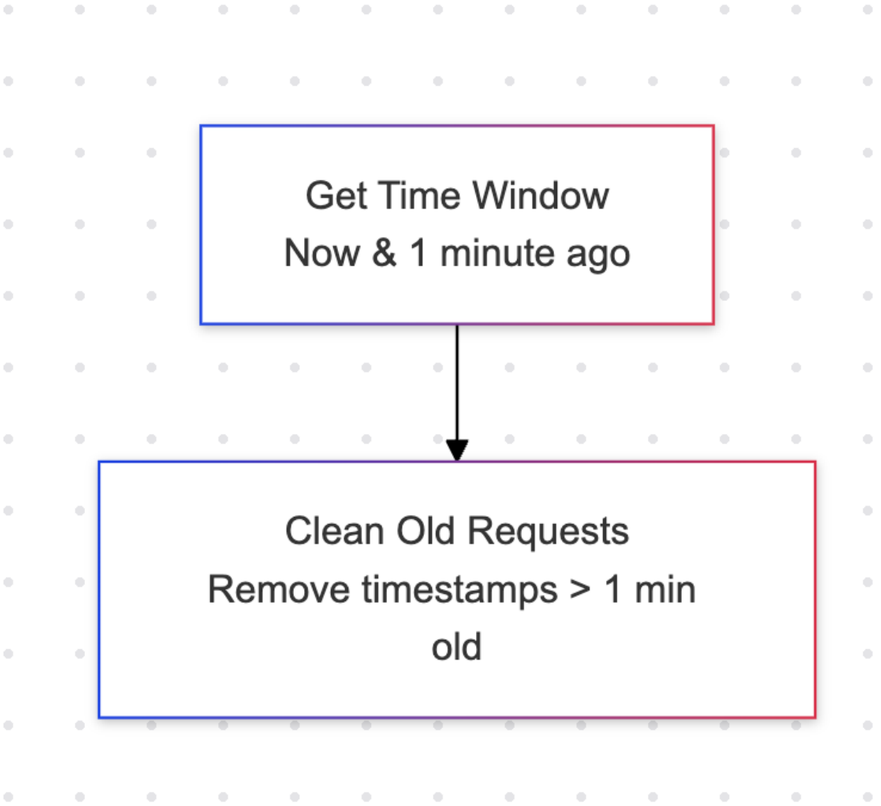
The rate limiter logic

```python
from datetime import datetime, timedelta

class RateLimiter:
    def __init__(self, requests_per_min: int = 10):
        self.requests_per_min = requests_per_min
        self.requests = defaultdict(list)
    def is_rate_limited(
        self, api_key: str
    ) -> tuple[bool, int]:
```



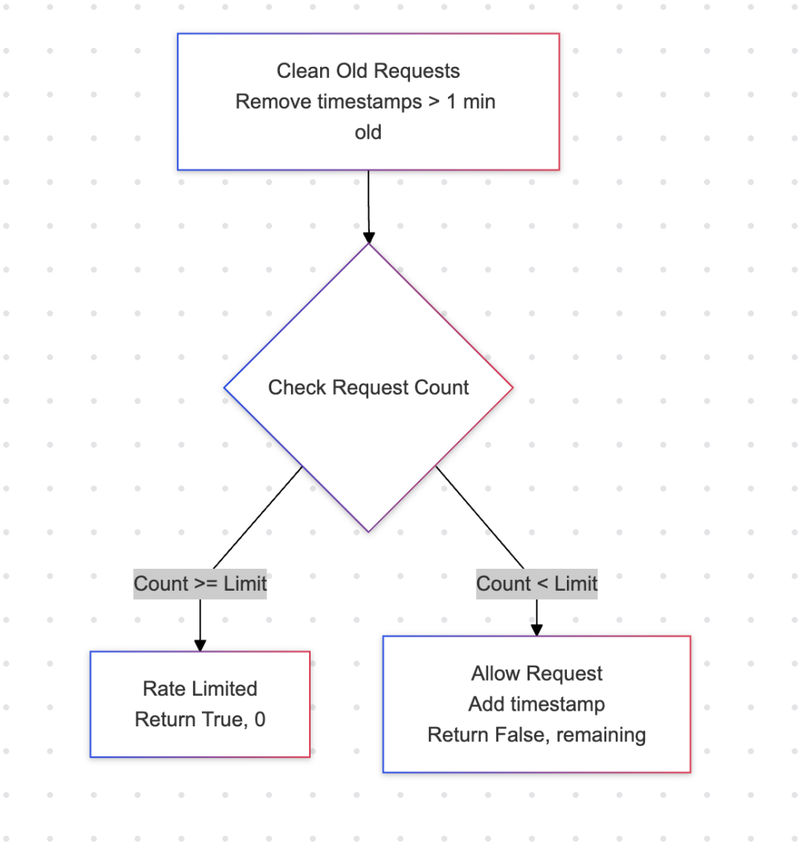
Deleting old requests

```python
from datetime import datetime, timedelta

class RateLimiter:
    def __init__(self, requests_per_min: int = 10):
        self.requests_per_min = requests_per_min
        self.requests = defaultdict(list)
    def is_rate_limited(
        self, api_key: str
    ) -> tuple[bool, int]:
        now = datetime.now()
        minute_ago = now - timedelta(minutes=1)
        self.requests[api_key] = [
            req_time for req_time in
          self.requests[api_key]
          if req_time > minute_ago
        ]
```



Check request count

```
    def is_rate_limited(self, api_key: str) ->
tuple[bool, int]:
        now = datetime.now()
        minute_ago = now - timedelta(minutes=1)
        self.requests[api_key] = [
          req_time for req_time in
          self.requests[api_key]
          if req_time > minute_ago
        ]
        recent_requests = len(self.requests[api_key])
        if recent_requests >= self.requests_per_min:
            return True, 0

        self.requests[api_key].append(now)
        return False
```



Add rate limit check

```
rate_limiter = RateLimiter(requests_per_minute=10)
def test_api_key(api_key: str = Depends(API_KEY_HEADER)):
    if api_key != API_KEY:
        raise HTTPException(
            status_code=403,
            detail="Invalid API key"
        )
    is_limited, _ = rate_limiter.is_rate_limited(api_key)
    if is_limited:
        raise HTTPException(
            status_code=429,
            detail="Rate limit exceeded. Please try again later."
        )
    return api_key
```

Apply rate limit to endpoint

```python
@app.post("/predict")
def predict_sentiment(
    request: SentimentRequest,
    api_key: str = Depends(test_api_key)
):
    result = sentiment_model(request.text)

    _, requests_remaining =
           rate_limiter.is_rate_limited(api_key)

    return {
        "text": request.text,
        "sentiment": result[0]["label"].lower(),
        "confidence": result[0]["score"],
        "requests_remaining": requests_remaining
    }
```

Send request 11 times:

```
curl -X POST "http://localhost:8000/predict" \
     -H "Content-Type: application/json" \
     -H "X-API-Key: your-secret-key" \
     -d '{"text": "I love this product"}'
```

Output:

```
{"detail":"Rate limit exceeded.
           Please try again later."}
```

Let's practice!

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

Asynchronous processing

DEPLOYING AI INTO PRODUCTION WITH FASTAPI

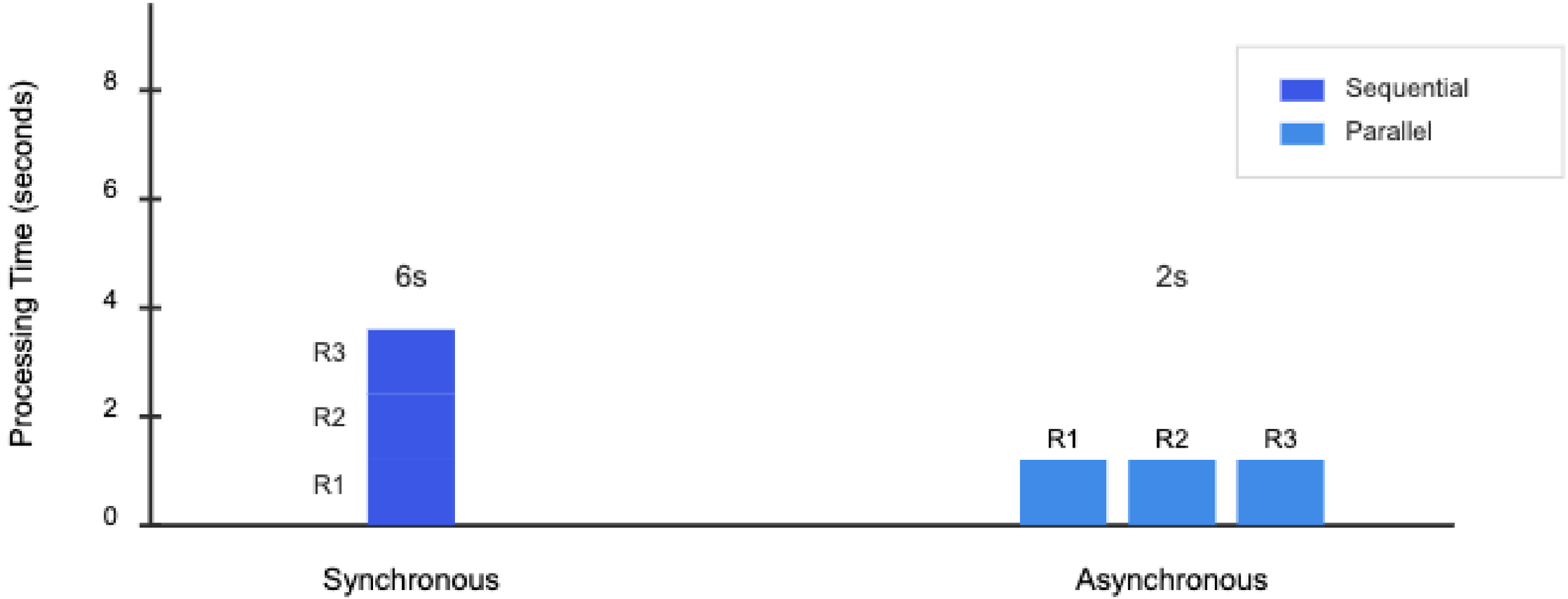
Matt Eckerle
Software and Data Engineering Leader

datacamp

What is asynchronous processing

Allows handling multiple requests concurrently

Synchronous vs asynchronous requests



Turning synchronous endpoints asynchronous

```python
@app.post("/analyze")
def analyze_sync(comment: Comment):
    result = sentiment_model(comment.text)
    return {"sentiment": result}
```

```python
import asyncio

@app.post("/analyze")
async def analyze_async(comment: Comment):
    result = await asyncio.to_thread(
        sentiment_model, comment.text
    )
    return {"sentiment": result}
```

Implementing background tasks

```
from fastapi import BackgroundTasks
from typing import List
```

```
@app.post("/analyze_batch")
async def analyze_batch(
    comments: Comments,
    background_tasks: BackgroundTasks
):
```

```
    async def process_comments(texts: List[str]):
        for text in texts:
            result = await asyncio.to_thread(
              sentiment_model, text)
    background_tasks.add_task(process_comments,
                              comments.texts)
    return {"message": "Processing started"}
```

- `BackgroundTasks` manage comment processing queue
- `background_tasks` handles post-response processing.
- `add_task` schedules `process_comments` asynchronously.

Adding error handling

```
@app.post("/analyze_comment")
async def analyze_comment(comment: Comment):
    try:
        sentiment_model = SentimentAnalyzer()
        result = await asyncio.wait_for(
            sentiment_model(comment.text),
            timeout=5.0
        )
        return {"sentiment": result["label"]}
```

Adding error handling

```
    except asyncio.TimeoutError:
        raise HTTPException(
            status_code=408,
            detail="Analysis timed out"
        )
```

```
    except Exception:
        raise HTTPException(
            status_code=500,
            detail="Analysis failed"
        )
```

Let's practice!

DEPLOYING AI INTO PRODUCTION WITH FASTAPI