GenAI: OpenAI API - Complete Tutorial

1. API Key Management

1.1 Understanding API Keys

```python
import os

# Set environment variable in your terminal first:
# export OPENAI_API_KEY="your-key-here"

api_key = os.environ.get('OPENAI_API_KEY')
if not api_key:
    raise ValueError("OPENAI_API_KEY environment variable not set")
```

1.2 Secure API Key Storage

Method 1: Environment Variables

```python
import os

# Load from environment
api_key = os.getenv('OPENAI_API_KEY')
org_id = os.getenv('OPENAI_ORG_ID')  # Optional
```

Method 2: .env Files

```bash
# Install python-dotenv
pip install python-dotenv
```

```
# .env file (in your project root)
OPENAI_API_KEY=sk-your-actual-key-here
OPENAI_ORG_ID=org-your-org-id
```

```python
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()

api_key = os.getenv('OPENAI_API_KEY')
org_id = os.getenv('OPENAI_ORG_ID')
```

1.3 .gitignore Configuration

```gitignore
# API Keys and Secrets
.env
.env.local
.env.development
.env.test
.env.production

# Python
__pycache__/
*.pyc
*.pyo

# API key files
api_keys.txt
secrets.json
config/secrets.py
```

1.4 Environment-Specific Configuration

python

```python
# dev_config.py
import os
from dotenv import load_dotenv

load_dotenv('.env.development')

class DevConfig:
    OPENAI_API_KEY = os.getenv('OPENAI_API_KEY')
    MODEL = "gpt-3.5-turbo"  # Cheaper for testing
    MAX_TOKENS = 100
```

python

```python
# prod_config.py
import os

class ProdConfig:
    OPENAI_API_KEY = os.getenv('OPENAI_API_KEY')
    MODEL = "gpt-4"
    MAX_TOKENS = 1000
    TIMEOUT = 30
```

1.5 Key Rotation Manager

```python
import datetime
import logging

class APIKeyManager:
    def __init__(self):
        self.primary_key = os.getenv('OPENAI_API_KEY_PRIMARY')
        self.backup_key = os.getenv('OPENAI_API_KEY_BACKUP')
        self.key_created_date = os.getenv('KEY_CREATED_DATE')

    def should_rotate_key(self):
        """Rotate keys every 90 days"""
        if not self.key_created_date:
            return True

        created = datetime.datetime.fromisoformat(self.key_created_date)
        age = datetime.datetime.now() - created
        return age.days > 90

    def get_active_key(self):
        if self.should_rotate_key():
            logging.warning("API key should be rotated")
        return self.primary_key
```

2. Python SDK

2.1 Installation and Setup

```bash
# Install the official OpenAI Python library
pip install openai

# For development, also install:
pip install python-dotenv
pip install requests
```

```python
import openai
print(f"OpenAI library version: {openai.__version__}")
```

2.2 Client Setup

```python
from openai import OpenAI
import os

# Initialize the client
client = OpenAI(
    api_key=os.getenv('OPENAI_API_KEY'),
    organization=os.getenv('OPENAI_ORG_ID')  # Optional
)

# Test authentication
try:
    models = client.models.list()
    print("Authentication successful!")
    print(f"Available models: {len(models.data)}")
except Exception as e:
    print(f"Authentication failed: {e}")
```

2.3 Advanced Client Configuration

```python
from openai import OpenAI
import httpx

# Client with custom settings
client = OpenAI(
    api_key=os.getenv('OPENAI_API_KEY'),
    timeout=httpx.Timeout(60.0, connect=10.0),
    max_retries=3,
    default_headers={"Custom-Header": "MyApp/1.0"}
)
```

2.4 Basic API Call

```python
def simple_chat_call():
    try:
        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[
                {"role": "user", "content": "Hello, how are you?"}
            ]
        )

        # Extract the response text
        answer = response.choices[0].message.content
        return answer

    except Exception as e:
        print(f"API call failed: {e}")
        return None

# Usage
result = simple_chat_call()
print(result)
```

2.5 Understanding Parameters

Model Selection

```python
def choose_model_example():
    # For creative writing
    response = client.chat.completions.create(
        model="gpt-4",  # Best creativity and reasoning
        messages=[{"role": "user", "content": "Write a short story"}]
    )

    # For simple Q&A
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",  # Faster, cheaper
        messages=[{"role": "user", "content": "What is 2+2?"}]
    )
```

Temperature Examples

python

```python
def temperature_examples():
    prompt = "Complete this sentence: The weather today is"

    # Low temperature = deterministic, focused
    conservative_response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": prompt}],
        temperature=0.1  # Very consistent
    )

    # Medium temperature = balanced
    balanced_response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": prompt}],
        temperature=0.7  # Good mix
    )

    # High temperature = creative, random
    creative_response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": prompt}],
        temperature=1.5  # Very creative
    )
```

Max Tokens Control

python

```python
def token_limit_examples():
    # Short responses
    brief_response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": "Explain Python in one sentence"}],
        max_tokens=50
    )

    # Long responses
    detailed_response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": "Explain Python programming"}],
        max_tokens=500
    )
```

2.6 Conversation Management

```python
def conversation_example():
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[
            {"role": "system", "content": "You are a helpful assistant"},
            {"role": "user", "content": "Explain recursion"},
            {"role": "assistant", "content": "Recursion is when a function calls itself..."},
            {"role": "user", "content": "Give me an example"}
        ]
    )
    return response.choices[0].message.content
```

3. Prompt Engineering

3.1 Understanding Roles

```python
# System role examples
system_msg_consultant = """
You are a senior Python developer with 10 years of experience.
Always provide code examples and explain the reasoning.
Focus on best practices, performance, and maintainability.
"""

system_msg_teacher = """
You are teaching programming to complete beginners.
Use simple language, analogies, and step-by-step explanations.
Always check if the user understands before moving to advanced topics.
"""
```

3.2 Prompt Patterns

Instruction-Based Prompting

```python
def instruction_based_prompts():
    # Weak instruction
    weak = "Make this better: def add(a,b): return a+b"

    # Strong instruction
    strong = """
    Improve this Python function following these requirements:
    1. Add type hints
    2. Add docstring with example
    3. Add input validation
    4. Handle edge cases

    Original function:
    def add(a,b): return a+b

    Return only the improved code with comments explaining changes.
    """
```

Role-Based Prompting

```python
def role_based_examples():
    code_reviewer = {
        "role": "system",
        "content": """
        You are a senior code reviewer at a top tech company.
        Review code for:
        - Security vulnerabilities
        - Performance issues
        - Code style and readability
        - Best practices violations

        Provide specific, actionable feedback with severity levels.
        """
    }
```

Few-Shot Prompting

```python
def few_shot_prompting():
    few_shot_messages = [
        {"role": "system", "content": "Convert natural language to SQL queries"},
        {"role": "user", "content": "Show me all customers from New York"},
        {"role": "assistant", "content": "SELECT * FROM customers WHERE city = 'New York';"},
        {"role": "user", "content": "Find the average order value by customer"},
        {"role": "assistant", "content": """
        SELECT customer_id, AVG(order_value) as avg_order_value
        FROM orders
        GROUP BY customer_id;
        """},
        {"role": "user", "content": "Get the top 5 products by sales this month"}
    ]
```

Chain-of-Thought Prompting

```python
def chain_of_thought_example():
    cot_prompt = """
    Solve this step by step:

    A Python list has 1000 elements. I want to find all elements that are:
    1. Even numbers
    2. Greater than 100
    3. Divisible by 3

    Think through this problem:
    1. What's the most efficient approach?
    2. What Python features should I use?
    3. How can I make it readable?
    4. Write the code with explanation

    Walk me through your reasoning for each step.
    """
```

3.3 Prompt Optimization

Clarity and Specificity

```python
# Vague prompt
vague = "Help me with my Python code"

# Specific prompt
specific = """
I have a Python function that processes CSV files, but it's running slowly on large files (>100MB).

Current code:
```python
def process_csv(filename):
    df = pd.read_csv(filename)
    return df.groupby('category').sum()
```

Please suggest 3 specific optimizations for handling large CSV files efficiently.
Include code examples and explain the performance benefits of each approach.
"""
```

```
#### Adding Constraints

```python
constrained_prompt = """
Explain Python decorators.

Constraints:
- Maximum 200 words
- Include exactly 1 code example
- Use beginner-friendly language
- End with a practical use case
- Format as numbered steps
"""
```
```

Iterative Improvement

```python
# Version 1 - Basic
v1 = "Write a function to validate email addresses"

# Version 2 - More specific
v2 = """
Write a Python function to validate email addresses.
Should return True/False and handle common invalid formats.
"""

# Version 3 - Complete specification
v3 = """
Create a Python email validation function with these requirements:

Function signature: validate_email(email: str) -> tuple[bool, str]

Returns:
- (True, "Valid") for valid emails
- (False, reason) for invalid emails

Validation rules:
- Must contain exactly one @
- Local part (before @): 1-64 characters, alphanumeric + ._-
- Domain part: valid domain format
- No consecutive dots

Include docstring and 3 test cases (valid, invalid format, edge case).
"""
```

3.4 Common Pitfalls

Avoid Vague Prompts

```python
# Bad - Too vague
bad_prompt = "Fix my code"

# Good - Specific
good_prompt = """
My Python function has a bug - it should remove duplicates from a list while preserving order, but it's not working corr

Current code:
```python
def remove_duplicates(lst):
    return list(set(lst))
```

Expected: [1, 2, 3, 4]
Actual: [1, 3, 2, 4] (order not preserved)

Please fix the function and explain why the original approach failed.
"""
```

```
#### Avoid Contradictory Instructions

```python
# Bad - Contradictory
contradictory = """
Write a very detailed explanation of Python classes.
Keep it brief and under 50 words.
Make it comprehensive but concise.
"""

# Good - Clear
clear_instruction = """
Write a concise explanation of Python classes (100-150 words).
Include:
- One simple class definition example
- Brief explanation of __init__ method
- One example of creating an instance

Target audience: Python beginners who understand functions.
"""
```

4. Handling the Response

4.1 Response Structure Analysis
```

```python
def analyze_response_structure():
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[{"role": "user", "content": "Hello!"}]
    )

    # Key components
    print("ID:", response.id)
    print("Model used:", response.model)
    print("Created timestamp:", response.created)
    print("Usage stats:", response.usage)
    print("Choices count:", len(response.choices))

    # The actual message
    choice = response.choices[0]
    print("Message role:", choice.message.role)
    print("Message content:", choice.message.content)
    print("Finish reason:", choice.finish_reason)
```

4.2 Safe Content Extraction

```python
def extract_content_safely():
    try:
        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[{"role": "user", "content": "What is Python?"}]
        )

        # Safe extraction with error checking
        if response.choices and len(response.choices) > 0:
            message = response.choices[0].message
            if message and message.content:
                return message.content.strip()
            else:
                return "No content in response"
        else:
            return "No choices in response"

    except Exception as e:
        return f"Error extracting content: {e}"
```

4.3 Advanced Content Processing

```python
def process_response_content(response):
    """Process and clean response content"""
    if not response.choices:
        return None, "No response choices"

    choice = response.choices[0]
    content = choice.message.content

    if not content:
        return None, f"Empty content, finish reason: {choice.finish_reason}"

    # Clean and process content
    processed_content = content.strip()

    # Check for completion issues
    finish_reason = choice.finish_reason
    warnings = []

    if finish_reason == "length":
        warnings.append("Response was cut off due to max_tokens limit")
    elif finish_reason == "content_filter":
        warnings.append("Response was filtered due to content policy")

    return processed_content, warnings
```

4.4 Comprehensive Error Handling

python

```python
import time
from openai import OpenAI, RateLimitError, APITimeoutError, APIError

def robust_api_call(messages, max_retries=3, retry_delay=1):
    """Make API call with comprehensive error handling"""

    for attempt in range(max_retries):
        try:
            response = client.chat.completions.create(
                model="gpt-3.5-turbo",
                messages=messages,
                timeout=30
            )

            return response, None

        except RateLimitError as e:
            error_msg = f"Rate limit exceeded: {e}"
            if attempt < max_retries - 1:
                wait_time = retry_delay * (2 ** attempt)  # Exponential backoff
                print(f"Rate limited, waiting {wait_time} seconds...")
                time.sleep(wait_time)
                continue
            return None, error_msg

        except APITimeoutError as e:
            error_msg = f"Request timed out: {e}"
            if attempt < max_retries - 1:
                print(f"Timeout, retrying attempt {attempt + 2}...")
                time.sleep(retry_delay)
                continue
            return None, error_msg

        except APIError as e:
            # Server errors (5xx) - retry
            if hasattr(e, 'status_code') and 500 <= e.status_code < 600:
                if attempt < max_retries - 1:
                    print(f"Server error {e.status_code}, retrying...")
                    time.sleep(retry_delay)
                    continue
            return None, f"API Error: {e}"

        except Exception as e:
            return None, f"Unexpected error: {e}"
```

```
    return None, "Max retries exceeded"
```

4.5 JSON Mode Response Handling

```python
def get_structured_response():
    """Request JSON response from the API"""
    response = client.chat.completions.create(
        model="gpt-3.5-turbo-1106",  # Supports JSON mode
        messages=[
            {"role": "system", "content": "You are a helpful assistant designed to output JSON."},
            {"role": "user", "content": """
            Analyze this sentence: "The quick brown fox jumps over the lazy dog"

            Return a JSON object with:
            - word_count: number of words
            - longest_word: the longest word
            - vowel_count: total vowels
            - sentiment: positive/negative/neutral
            """}
        ],
        response_format={"type": "json_object"}
    )

    # Parse JSON response
    try:
        import json
        content = response.choices[0].message.content
        data = json.loads(content)
        return data, None
    except json.JSONDecodeError as e:
        return None, f"Failed to parse JSON: {e}"
```

4.6 Response Validation

python

```python
import json
from typing import Dict, Any, Optional

def validate_json_response(response_content: str, expected_schema: Dict) -> tuple[Optional[Dict], Optional[str]]:
    """Validate JSON response against expected schema"""
    try:
        # Parse JSON
        data = json.loads(response_content)

        # Basic schema validation
        for key, expected_type in expected_schema.items():
            if key not in data:
                return None, f"Missing required field: {key}"

            if not isinstance(data[key], expected_type):
                return None, f"Field {key} should be {expected_type.__name__}, got {type(data[key]).__name__}"

        return data, None

    except json.JSONDecodeError as e:
        return None, f"Invalid JSON: {e}"

# Usage example
def get_validated_analysis():
    response = client.chat.completions.create(
        model="gpt-3.5-turbo-1106",
        messages=[
            {"role": "system", "content": "Output valid JSON only."},
            {"role": "user", "content": "Analyze 'Hello World' and return JSON with word_count (int) and uppercase_version (s
        ],
        response_format={"type": "json_object"}
    )

    # Define expected schema
    schema = {
        "word_count": int,
        "uppercase_version": str
    }

    content = response.choices[0].message.content
    data, error = validate_json_response(content, schema)

    if error:
        print(f"Validation error: {error}")
```

```
        return None

    return data
```

4.7 Token Usage Monitoring

python

```python
class TokenUsageTracker:
    def __init__(self):
        self.total_prompt_tokens = 0
        self.total_completion_tokens = 0
        self.total_requests = 0

    def track_response(self, response):
        """Track token usage from API response"""
        if hasattr(response, 'usage') and response.usage:
            self.total_prompt_tokens += response.usage.prompt_tokens
            self.total_completion_tokens += response.usage.completion_tokens
            self.total_requests += 1

            print(f"Request tokens: {response.usage.prompt_tokens}")
            print(f"Response tokens: {response.usage.completion_tokens}")
            print(f"Total tokens this request: {response.usage.total_tokens}")

    def get_summary(self):
        """Get usage summary"""
        total_tokens = self.total_prompt_tokens + self.total_completion_tokens
        return {
            "total_requests": self.total_requests,
            "total_prompt_tokens": self.total_prompt_tokens,
            "total_completion_tokens": self.total_completion_tokens,
            "total_tokens": total_tokens,
            "avg_tokens_per_request": total_tokens / max(self.total_requests, 1)
        }

# Usage
tracker = TokenUsageTracker()

response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[{"role": "user", "content": "Explain Python lists"}]
)

tracker.track_response(response)
summary = tracker.get_summary()
print(f"Total usage: {summary}")
```

4.8 Response Caching

python

```python
import hashlib
import json
import time
from typing import Optional

class ResponseCache:
    def __init__(self, ttl_seconds=3600):  # 1 hour TTL
        self.cache = {}
        self.ttl = ttl_seconds

    def _generate_key(self, messages, model, **kwargs):
        """Generate cache key from request parameters"""
        cache_data = {
            "messages": messages,
            "model": model,
            **kwargs
        }
        cache_string = json.dumps(cache_data, sort_keys=True)
        return hashlib.md5(cache_string.encode()).hexdigest()

    def get(self, messages, model, **kwargs) -> Optional[str]:
        """Get cached response if available and not expired"""
        key = self._generate_key(messages, model, **kwargs)

        if key in self.cache:
            cached_data = self.cache[key]
            if time.time() - cached_data['timestamp'] < self.ttl:
                print("Using cached response")
                return cached_data['response']
            else:
                # Remove expired entry
                del self.cache[key]

        return None

    def set(self, messages, model, response_content, **kwargs):
        """Cache response"""
        key = self._generate_key(messages, model, **kwargs)
        self.cache[key] = {
            'response': response_content,
            'timestamp': time.time()
        }
        print("Response cached")
```

```python
def cached_api_call(messages, model="gpt-3.5-turbo", **kwargs):
    # Try to get from cache first
    cached_response = cache.get(messages, model, **kwargs)
    if cached_response:
        return cached_response, True  # True indicates cached

    # Make API call
    try:
        response = client.chat.completions.create(
            model=model,
            messages=messages,
            **kwargs
        )

        content = response.choices[0].message.content

        # Cache the response
        cache.set(messages, model, content, **kwargs)

        return content, False  # False indicates fresh API call

    except Exception as e:
        return f"Error: {e}", False

# Usage
cache = ResponseCache(ttl_seconds=1800)  # 30 minutes
messages = [{"role": "user", "content": "What is machine learning?"}]

response1, was_cached = cached_api_call(messages)
print(f"First call - Cached: {was_cached}")

response2, was_cached = cached_api_call(messages)
print(f"Second call - Cached: {was_cached}")
```

5. Production-Ready Response Handler

python

```python
import logging
import time
import json
from typing import Optional, Dict, List, Tuple, Any
from openai import OpenAI, RateLimitError, APITimeoutError, APIError

class OpenAIResponseHandler:
    def __init__(self, client: OpenAI, max_retries=3, base_delay=1):
        self.client = client
        self.max_retries = max_retries
        self.base_delay = base_delay
        self.logger = logging.getLogger(__name__)

        # Setup logging
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        )

    def make_request(self,
                     messages: List[Dict[str, str]],
                     model: str = "gpt-3.5-turbo",
                     temperature: float = 0.7,
                     max_tokens: Optional[int] = None,
                     json_mode: bool = False,
                     **kwargs) -> Tuple[Optional[str], Optional[Dict[str, Any]], Optional[str]]:
        """
        Make API request with comprehensive error handling

        Returns:
            (content, metadata, error_message)
        """

        # Prepare request parameters
        request_params = {
            "model": model,
            "messages": messages,
            "temperature": temperature,
            **kwargs
        }

        if max_tokens:
            request_params["max_tokens"] = max_tokens

        if
```

```
if json_mode:
    request_params["response_format"] = {"type": "json_object"}

# Attempt request with retries
for attempt in range(self.max_retries):
    try:
        self.logger.info(f"Making API request (attempt {attempt + 1}/{self.max_retries})")

        response = self.client.chat.completions.create(**request_params)

        # Process successful response
        content, metadata, error = self._process_response(response, json_mode)

        if error:
            self.logger.error(f"Response processing error: {error}")
            return None, None, error

        self.logger.info("API request successful")
        return content, metadata, None

    except RateLimitError as e:
        error_msg = f"Rate limit exceeded: {e}"
        self.logger.warning(error_msg)

        if attempt < self.max_retries - 1:
            wait_time = self.base_delay * (2 ** attempt)
            self.logger.info(f"Waiting {wait_time} seconds before retry...")
            time.sleep(wait_time)
            continue
        return None, None, error_msg

    except APITimeoutError as e:
        error_msg = f"Request timed out: {e}"
        self.logger.warning(error_msg)

        if attempt < self.max_retries - 1:
            self.logger.info("Retrying after timeout...")
            time.sleep(self.base_delay)
            continue
        return None, None, error_msg

    except APIError as e:
        error_msg = f"API Error: {e}"
        self.logger.error(error_msg)

        # Retry on server errors (5xx)
        if hasattr(e, 'status_code') and 500 <= e.status_code < 600:
```

```
            if attempt < self.max_retries - 1:
                self.logger.info("Retrying after server error...")
                time.sleep(self.base_delay)
                continue

        return None, None, error_msg

    except Exception as e:
        error_msg = f"Unexpected error: {e}"
        self.logger.error(error_msg)
        return None, None, error_msg

return None, None, "Max retries exceeded"

def _process_response(self, response, json_mode=False) -> Tuple[Optional[str], Dict[str, Any], Optional[str]]:
    """Process API response and extract information"""

    try:
        # Validate response structure
        if not response.choices or len(response.choices) == 0:
            return None, {}, "No choices in response"

        choice = response.choices[0]
        content = choice.message.content

        if not content:
            return None, {}, f"Empty content. Finish reason: {choice.finish_reason}"

        # Prepare metadata
        metadata = {
            "model": response.model,
            "finish_reason": choice.finish_reason,
            "created": response.created,
            "response_id": response.id
        }

        # Add usage information if available
        if hasattr(response, 'usage') and response.usage:
            metadata["usage"] = {
                "prompt_tokens": response.usage.prompt_tokens,
                "completion_tokens": response.usage.completion_tokens,
                "total_tokens": response.usage.total_tokens
            }

        # Handle JSON mode
        if json_mode:
```

```
            try:
                parsed_content = json.loads(content)
                metadata["json_parsed"] = True
                return parsed_content, metadata, None
            except json.JSONDecodeError as e:
                return None, metadata, f"Failed to parse JSON response: {e}"

        # Check for truncation
        if choice.finish_reason == "length":
            metadata["warning"] = "Response was truncated due to token limit"
        elif choice.finish_reason == "content_filter":
            return None, metadata, "Response was filtered due to content policy"

        return content.strip(), metadata, None

    except Exception as e:
        return None, {}, f"Error processing response: {e}"

def chat(self, user_message: str, system_message: Optional[str] = None, **kwargs) -> Tuple[Optional[str], Optional[str
    """Simplified chat interface"""

    messages = []
    if system_message:
        messages.append({"role": "system", "content": system_message})
    messages.append({"role": "user", "content": user_message})

    content, metadata, error = self.make_request(messages, **kwargs)

    if error:
        return None, error

    # Log usage if available
    if metadata and "usage" in metadata:
        usage = metadata["usage"]
        self.logger.info(f"Token usage - Prompt: {usage['prompt_tokens']}, "
                         f"Completion: {usage['completion_tokens']}, "
                         f"Total: {usage['total_tokens']}")

    return content, None

# Usage Examples
def main():
    # Initialize handler
    client = OpenAI(api_key=os.getenv('OPENAI_API_KEY'))
    handler = OpenAIResponseHandler(client)

    # Simple chat
```

```
    # Simple chat
    response, error = handler.chat(
        user_message="Explain Python list comprehensions in simple terms",
        system_message="You are a helpful Python tutor",
        temperature=0.3
    )

    if error:
        print(f"Error: {error}")
    else:
        print(f"Response: {response}")

    # JSON mode example
    json_response, metadata, error = handler.make_request(
        messages=[
            {"role": "system", "content": "You are a data analyzer. Always respond with valid JSON."},
            {"role": "user", "content": """
            Analyze this text: "Python is awesome for data science"

            Return JSON with:
            - word_count: int
            - sentiment: string (positive/negative/neutral)
            - main_topic: string
            """}
        ],
        json_mode=True,
        temperature=0.1
    )

    if error:
        print(f"JSON Error: {error}")
    else:
        print(f"JSON Response: {json_response}")
        print(f"Metadata: {metadata}")

if __name__ == "__main__":
    main()
```

6. Complete Working Examples

6.1 Simple Chat Bot

python

```python
from openai import OpenAI
import os
from dotenv import load_dotenv

load_dotenv()

class SimpleChatBot:
    def __init__(self):
        self.client = OpenAI(api_key=os.getenv('OPENAI_API_KEY'))
        self.conversation_history = []

    def chat(self, user_input: str, system_prompt: str = None) -> str:
        # Add system message if provided and not already present
        if system_prompt and not any(msg.get('role') == 'system' for msg in self.conversation_history):
            self.conversation_history.insert(0, {"role": "system", "content": system_prompt})

        # Add user message
        self.conversation_history.append({"role": "user", "content": user_input})

        try:
            response = self.client.chat.completions.create(
                model="gpt-3.5-turbo",
                messages=self.conversation_history,
                temperature=0.7,
                max_tokens=500
            )

            assistant_response = response.choices[0].message.content

            # Add assistant response to history
            self.conversation_history.append({"role": "assistant", "content": assistant_response})

            return assistant_response

        except Exception as e:
            return f"Error: {e}"

    def clear_history(self):
        self.conversation_history = []

# Usage
bot = SimpleChatBot()

# Set system context
```

```python
system_msg = "You are a helpful Python programming tutor. Keep responses concise and practical."

print("Python Tutor Bot (type 'quit' to exit, 'clear' to reset)")
while True:
    user_input = input("\nYou: ")

    if user_input.lower() == 'quit':
        break
    elif user_input.lower() == 'clear':
        bot.clear_history()
        print("Conversation cleared!")
        continue

    response = bot.chat(user_input, system_msg)
    print(f"\nBot: {response}")
```

6.2 Code Review Assistant

python

```python
class CodeReviewAssistant:
    def __init__(self):
        self.client = OpenAI(api_key=os.getenv('OPENAI_API_KEY'))
        self.system_prompt = """
        You are a senior software engineer conducting code reviews.

        For each code review:
        1. Analyze code quality, performance, and best practices
        2. Identify potential bugs or security issues
        3. Suggest improvements with explanations
        4. Rate the code from 1-10
        5. Provide specific, actionable feedback

        Format your response as:
        ## Overall Rating: X/10
        ## Issues Found:
        ## Suggestions:
        ## Improved Code:
        """

    def review_code(self, code: str, language: str = "Python") -> str:
        prompt = f"""
        Please review this {language} code:

        ```{language.lower()}
        {code}
        ```
        """

        try:
            response = self.client.chat.completions.create(
                model="gpt-4", # Use GPT-4 for better code analysis
                messages=[
                    {"role": "system", "content": self.system_prompt},
                    {"role": "user", "content": prompt}
                ],
                temperature=0.3 # Lower temperature for more consistent reviews
)

            return response.choices[0].message.content

        except Exception as e:
            return f"Code review failed: {e}"
```

```python
# Usage
reviewer = CodeReviewAssistant()

code_to_review = '''
def calculate_total(items):
    total = 0
    for i in range(len(items)):
        total = total + items[i]['price'] * items[i]['quantity']
    return total

items = [{'price': 10, 'quantity': 2}, {'price': 15, 'quantity': 1}]
print(calculate_total(items))
'''

review = reviewer.review_code(code_to_review)
print(review)
```

6.3 Structured Data Extractor

python

```python
import json
from typing import Dict, List, Optional

class DataExtractor:
    def __init__(self):
        self.client = OpenAI(api_key=os.getenv('OPENAI_API_KEY'))

    def extract_structured_data(self, text: str, schema: Dict) -> Optional[Dict]:
        """Extract structured data from text based on provided schema"""

        # Create schema description
        schema_description = "Extract the following information:\n"
        for field, field_type in schema.items():
            schema_description += f"- {field}: {field_type.__name__}\n"

        prompt = f"""
        {schema_description}

        Text to analyze:
        "{text}"

        Return the extracted data as valid JSON only.
        """

        try:
            response = self.client.chat.completions.create(
                model="gpt-3.5-turbo-1106",
                messages=[
                    {"role": "system", "content": "You are a data extraction assistant. Always return valid JSON."},
                    {"role": "user", "content": prompt}
                ],
                response_format={"type": "json_object"},
                temperature=0.1
            )

            extracted_data = json.loads(response.choices[0].message.content)

            # Validate against schema
            for field, expected_type in schema.items():
                if field not in extracted_data:
                    print(f"Warning: Missing field '{field}'")
                elif not isinstance(extracted_data[field], expected_type):
                    print(f"Warning: Field '{field}' has incorrect type")
```

```
        return extracted_data

    except Exception as e:
        print(f"Extraction failed: {e}")
        return None

# Usage
extractor = DataExtractor()

# Define what we want to extract
schema = {
    "name": str,
    "email": str,
    "phone": str,
    "company": str,
    "job_title": str
}

# Sample text
text = """
Hi, my name is John Smith and I work as a Senior Developer at TechCorp.
You can reach me at john.smith@techcorp.com or call me at (555) 123-4567.
"""

result = extractor.extract_structured_data(text, schema)
if result:
    print("Extracted data:")
    for key, value in result.items():
        print(f"  {key}: {value}")
```

6.4 Batch Processing Manager

python

```python
import time
from typing import List, Dict, Callable
from concurrent.futures import ThreadPoolExecutor
import threading

class BatchProcessor:
    def __init__(self, max_workers=3, rate_limit_per_minute=60):
        self.client = OpenAI(api_key=os.getenv('OPENAI_API_KEY'))
        self.max_workers = max_workers
        self.rate_limit = rate_limit_per_minute
        self.request_times = []
        self.lock = threading.Lock()

    def _check_rate_limit(self):
        """Implement rate limiting"""
        with self.lock:
            current_time = time.time()
            # Remove requests older than 1 minute
            self.request_times = [t for t in self.request_times if current_time - t < 60]

            if len(self.request_times) >= self.rate_limit:
                sleep_time = 60 - (current_time - self.request_times[0])
                if sleep_time > 0:
                    print(f"Rate limit reached, sleeping for {sleep_time:.2f} seconds")
                    time.sleep(sleep_time)

            self.request_times.append(current_time)

    def process_single_item(self, item: Dict) -> Dict:
        """Process a single item"""
        self._check_rate_limit()

        try:
            response = self.client.chat.completions.create(
                model=item.get('model', 'gpt-3.5-turbo'),
                messages=item['messages'],
                temperature=item.get('temperature', 0.7),
                max_tokens=item.get('max_tokens', 500)
            )

            return {
                'id': item.get('id', 'unknown'),
                'success': True,
                'response': response.choices[0].message.content,
```

```
            'usage': response.usage.total_tokens if response.usage else None
        }

    except Exception as e:
        return {
            'id': item.get('id', 'unknown'),
            'success': False,
            'error': str(e),
            'response': None
        }

def process_batch(self, items: List[Dict]) -> List[Dict]:
    """Process multiple items in parallel"""
    print(f"Processing {len(items)} items with {self.max_workers} workers")

    results = []
    with ThreadPoolExecutor(max_workers=self.max_workers) as executor:
        # Submit all tasks
        future_to_item = {executor.submit(self.process_single_item, item): item for item in items}

        # Collect results as they complete
        for future in future_to_item:
            try:
                result = future.result()
                results.append(result)

                if result['success']:
                    print(f"✓ Completed item {result['id']}")
                else:
                    print(f"✗ Failed item {result['id']}: {result['error']}")

            except Exception as e:
                item = future_to_item[future]
                results.append({
                    'id': item.get('id', 'unknown'),
                    'success': False,
                    'error': f"Future exception: {e}",
                    'response': None
                })

    return results

# Usage
processor = BatchProcessor(max_workers=2, rate_limit_per_minute=30)

# Prepare batch items
batch_items = [
```

```
    {
        'id': 'task_1',
        'messages': [{'role': 'user', 'content': 'Explain Python lists in 2 sentences'}],
        'temperature': 0.3
    },
    {
        'id': 'task_2',
        'messages': [{'role': 'user', 'content': 'What are Python dictionaries?'}],
        'temperature': 0.3
    },
    {
        'id': 'task_3',
        'messages': [{'role': 'user', 'content': 'Explain Python functions briefly'}],
        'temperature': 0.3
    }
]

# Process batch
results = processor.process_batch(batch_items)

# Display results
for result in results:
    print(f"\n--- Task {result['id']} ---")
    if result['success']:
        print(f"Response: {result['response']}")
        print(f"Tokens used: {result['usage']}")
    else:
        print(f"Error: {result['error']}")
```

7. Best Practices Summary

7.1 Security Best Practices

```python
# ✅ DO: Use environment variables
api_key = os.getenv('OPENAI_API_KEY')

# ❌ DON'T: Hardcode API keys
api_key = "sk-proj-abc123..."  # Never do this!

# ✅ DO: Validate environment setup
if not os.getenv('OPENAI_API_KEY'):
    raise ValueError("OPENAI_API_KEY not found in environment variables")

# ✅ DO: Use .gitignore properly
"""
.env
*.key
secrets/
config/api_keys.py
"""
```

7.2 Error Handling Best Practices

```python
# ✅ DO: Implement retry logic
def make_api_call_with_retry(messages, max_retries=3):
    for attempt in range(max_retries):
        try:
            return client.chat.completions.create(
                model="gpt-3.5-turbo",
                messages=messages
            )
        except RateLimitError:
            if attempt < max_retries - 1:
                time.sleep(2 ** attempt)  # Exponential backoff
                continue
            raise
        except Exception as e:
            if attempt == max_retries - 1:
                raise
            time.sleep(1)

# ✅ DO: Handle specific exceptions
try:
    response = client.chat.completions.create(...)
except RateLimitError:
    print("Rate limited - wait before retrying")
except APITimeoutError:
    print("Request timed out - check connection")
except APIError as e:
    print(f"API error: {e}")
```

7.3 Cost Optimization Best Practices

```python
# ✅ DO: Monitor token usage
def track_usage(response):
    if response.usage:
        print(f"Tokens used: {response.usage.total_tokens}")
        print(f"Cost estimate: ${response.usage.total_tokens * 0.0015 / 1000:.4f}")

# ✅ DO: Use appropriate models
# For simple tasks
model = "gpt-3.5-turbo"  # Cheaper, faster

# For complex reasoning
model = "gpt-4"  # More expensive, better quality

# ✅ DO: Set reasonable token limits
response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=messages,
    max_tokens=200  # Prevent unexpectedly long responses
)

# ✅ DO: Use caching for repeated queries
cache = {}
def cached_request(messages_hash, messages):
    if messages_hash in cache:
        return cache[messages_hash]

    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=messages
    )

    cache[messages_hash] = response
    return response
```

7.4 Prompt Engineering Best Practices

```python
# ✅ DO: Be specific and clear
good_prompt = """
Convert this Python list to a dictionary where:
- Keys are the original list values
- Values are the index positions
- Input: [apple, banana, cherry]
- Expected output: {'apple': 0, 'banana': 1, 'cherry': 2}

Provide only the code solution.
"""

# ❌ DON'T: Be vague
bad_prompt = "Help me with lists and dictionaries"

# ✅ DO: Use system messages effectively
system_message = """
You are a Python code reviewer. For each code submission:
1. Check for bugs and logic errors
2. Suggest performance improvements
3. Ensure code follows PEP 8 style guide
4. Provide specific, actionable feedback
"""

# ✅ DO: Use appropriate temperature
# For factual/deterministic tasks
temperature = 0.1

# For creative tasks
temperature = 0.8

# For balanced responses
temperature = 0.7
```

7.5 Production Deployment Checklist

python

```python
# ✅ Production-ready configuration
class ProductionConfig:
    def __init__(self):
        self.api_key = os.getenv('OPENAI_API_KEY')
        self.max_retries = 3
        self.timeout = 30
        self.rate_limit = 60  # requests per minute

        # Validation
        if not self.api_key:
            raise ValueError("Missing OPENAI_API_KEY")

        # Logging setup
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(levelname)s - %(message)s',
            handlers=[
                logging.FileHandler('openai_api.log'),
                logging.StreamHandler()
            ]
        )

# ✅ Health check endpoint
def health_check():
    try:
        client.models.list()
        return {"status": "healthy", "timestamp": time.time()}
    except Exception as e:
        return {"status": "unhealthy", "error": str(e), "timestamp": time.time()}

# ✅ Monitoring and alerts
def log_api_metrics(response, start_time):
    duration = time.time() - start_time

    metrics = {
        "duration": duration,
        "model": response.model,
        "tokens_used": response.usage.total_tokens if response.usage else 0,
        "timestamp": time.time()
    }

    logging.info(f"API_METRICS: {json.dumps(metrics)}")

    # Alert if response takes too long
```

```
    if duration > 30:  # 30 seconds
        logging.warning(f"Slow API response: {duration:.2f}s")

# ✅ Circuit breaker pattern
class CircuitBreaker:
    def __init__(self, failure_threshold=5, recovery_timeout=60):
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.failure_count = 0
        self.last_failure_time = None
        self.state = 'closed'  # closed, open, half-open

    def call(self, func, *args, **kwargs):
        if self.state == 'open':
            if time.time() - self.last_failure_time > self.recovery_timeout:
                self.state = 'half-open'
            else:
                raise Exception("Circuit breaker is open")

        try:
            result = func(*args, **kwargs)
            self.failure_count = 0
            self.state = 'closed'
            return result
        except Exception as e:
            self.failure_count += 1
            self.last_failure_time = time.time()

            if self.failure_count >= self.failure_threshold:
                self.state = 'open'

            raise e
```

8. Common Troubleshooting

8.1 Authentication Issues

```python
# Problem: Invalid API key
# Solution: Verify key format and permissions
def verify_api_key():
    api_key = os.getenv('OPENAI_API_KEY')

    # Check format
    if not api_key or not api_key.startswith('sk-'):
        print("❌ Invalid API key format")
        return False

    # Test with simple request
    try:
        client = OpenAI(api_key=api_key)
        client.models.list()
        print("✅ API key is valid")
        return True
    except Exception as e:
        print(f"❌ API key validation failed: {e}")
        return False
```

8.2 Rate Limit Issues

```python
# Problem: Rate limit exceeded
# Solution: Implement exponential backoff
def handle_rate_limits():
    import random

    for attempt in range(5):
        try:
            response = client.chat.completions.create(...)
            return response
        except RateLimitError as e:
            if attempt == 4:  # Last attempt
                raise e

            # Exponential backoff with jitter
            wait_time = (2 ** attempt) + random.uniform(0, 1)
            print(f"Rate limited, waiting {wait_time:.2f} seconds...")
            time.sleep(wait_time)
```

8.3 Response Quality Issues

```python
# Problem: Inconsistent responses
# Solution: Optimize prompts and parameters
def improve_response_quality():
    # Use lower temperature for consistency
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[
            {"role": "system", "content": "You are a precise technical assistant."},
            {"role": "user", "content": "Explain Python classes"}
        ],
        temperature=0.3,  # Lower for consistency
        max_tokens=300,   # Reasonable limit
        top_p=0.9         # Focus on likely tokens
    )

    return response.choices[0].message.content
```

This comprehensive tutorial covers everything you need to know about working with the OpenAI API in Python, from basic setup to production-ready implementations. Copy and paste any section you need!