TL;DR

- The Responses API is the primary interface in the official OpenAI Python SDK for generating model outputs; Chat Completions remains supported indefinitely as a legacy alternative [S2].
- Minimal usage involves client.responses.create with model, input (string or structured parts), and optional instructions; output is conveniently accessible via response.output_text [S2].
- Vision is supported by mixing input_text and input_image content parts (via HTTP URL or base64 data URL) in the same request [S2].
- Streaming is available by passing stream=True and iterating SSE events; both sync and async patterns are supported in the Python SDK [S2].
- Use environment variables for OPENAI_API_KEY (e.g., via python-dotenv) and consider AsyncOpenAI and aiohttp for concurrency; requests/responses are typed (TypedDict/Pydantic) in the SDK [S2].

OpenAI Responses API basics

1) What the Responses API is

- The official Python SDK identifies the Responses API as the primary API for interacting with OpenAI models, supplanting Chat Completions as the default pattern while the Chat Completions API remains supported indefinitely [S2].
- Core (documented) capabilities in SDK examples:
- Text generation with response.output_text for convenience [S2].
- Multimodal vision input by including input_text and input_image parts together in input [S2].
- Streaming via Server-Sent Events when stream=True [S2].

2) Models and when to use them

- The SDK examples show model selection via a string (e.g., "gpt-5.2"); specific model families, modalities, reasoning capabilities, and token limits are not enumerated in the provided source [S2].
- Guidance on choosing among models (latency, cost, reasoning depth, modality needs) is not provided in the provided source [S2].

3) Authentication and setup

- Install the OpenAI Python library from PyPI and initialize the client with OpenAI(); it reads the API key from the OPENAI_API_KEY environment variable by default [S2].
- The SDK recommends storing the key in a .env file via python-dotenv rather than hardcoding it in source control [S2].
- Both synchronous (OpenAI) and asynchronous (AsyncOpenAI) clients are available; AsyncOpenAI can optionally use aiohttp for improved concurrency [S2].
- Specific HTTP headers, organization/project scoping, or server-only usage guidance are not documented in the provided source [S2].

4) Core request/response shape

- With the Python SDK, create a response via client.responses.create(model=..., input=..., instructions=...); stream=True enables SSE streaming [S2].
- The input field may be:
- A simple string (e.g., "Explain ...") [S2].
- A list with role and content parts; content parts may include:
- { "type": "input_text", "text": "..."} [S2].
- { "type": "input_image", "image_url": "<http or data URL>"} [S2].
- The response object exposes response.output_text for the generated text in non-streaming mode [S2].

• Additional fields such as tools, response_format/JSON schema, temperature/top_p, max tokens, usage, finish reasons, and HTTP endpoint path are not documented in the provided source [S2].

5) Basic text generation
• Minimal (sync) example in Python:
• client.responses.create(model="gpt-5.2", instructions="...", input="..."); print(response.output_text) [S2].
• The provided source does not document temperature/top_p or determinism controls (e.g., seed), nor strict JSON controls for Responses [S2].

6) Structured output (JSON Schema)
• The provided source does not document response_format with json_schema or a “strict” mode for Responses [S2].

7) Tool/function calling
• The provided source does not document tool/function calling for the Responses API; it does note that the Realtime API supports function calling over WebSocket, which is distinct from Responses [S2].
• Best-practice guidance (validation/security/idempotency) for tool calls in Responses is not provided in the provided source [S2].

8) Multimodal I/O
• Vision input:
• Use content parts with input_text and input_image; image_url may be an HTTP URL or a base64 data URL like data:image/png;base64,... [S2].
• Audio input/output for the Responses API is not documented in the provided source; Realtime mentions audio I/O but is a different API [S2].

9) Streaming responses
• Set stream=True when calling client.responses.create and iterate the returned stream to consume SSE events in both sync and async clients [S2].
• The provided source does not define specific event types for Responses streaming or how to reconstruct tool calls; examples simply iterate and print events [S2].

10) Conversation state and multi-turn patterns
• The provided source does not document multi-turn conversation state management for the Responses API (e.g., passing prior turns vs managed threads) [S2].

11) Errors, retries, and idempotency
• For the Realtime API (not Responses), errors are delivered as events, the connection remains open, and the SDK does not raise exceptions for those errors; client code must handle error events explicitly [S2].
• Error codes, retry strategies, and idempotency keys for the Responses API are not documented in the provided source [S2].

12) Performance, costs, and limits
• The provided source does not document token accounting, rate limits, throughput strategies, pricing, or prompt caching for the Responses API [S2].

13) Security, safety, and compliance

• Store and access the API key via environment variables (e.g., OPENAI_API_KEY) rather than committing it to source control; using python-dotenv is recommended to manage keys locally via .env [S2].
• Additional guidance regarding PII handling, safety controls, content moderation, auditability, or data use policies is not provided in the provided source [S2].

14) SDKs and minimal examples
• Python quick start (sync):
• from openai import OpenAI; client = OpenAI(); response = client.responses.create(model="gpt-5.2", instructions="...", input="..."); print(response.output_text) [S2].
• Python quick start (async):
• from openai import AsyncOpenAI; client = AsyncOpenAI(); response = await client.responses.create(...); print(response.output_text) [S2].
• Vision example:
• Provide input as a list with role="user" and content including input_text and input_image (URL or base64 data URL) [S2].
• Streaming example:
• stream = client.responses.create(..., stream=True); iterate events; identical interface for AsyncOpenAI [S2].
• Types and helpers:
• Requests use TypedDicts, responses are Pydantic models (with helpers like to_json/to_dict) to improve typing and editor assistance [S2].
• JavaScript and curl examples are not provided in the provided source [S2].

15) Best practices and common pitfalls
• Prefer the Responses API for new integrations; keep Chat Completions for legacy or existing code paths as it remains supported indefinitely [S2].
• Manage secrets via environment variables (OPENAI_API_KEY) and avoid committing keys; consider python-dotenv for local development [S2].
• Use AsyncOpenAI (and optionally aiohttp) for higher concurrency, and leverage the streaming interface for incremental output when appropriate [S2].
• For vision, ensure images are accessible via HTTP(S) or encoded as base64 data URLs in input_image.image_url [S2].
• Utilize typed requests and Pydantic responses for better validation and IDE autocomplete; serialize with to_json/to_dict when needed [S2].

16) Appendix: Reference cheatsheet
• Minimal text request (sync):
• response = client.responses.create(model="gpt-5.2", instructions="You are a helper.", input="Say hello."); print(response.output_text) [S2].
• Vision (HTTP URL):
• input=[{"role":"user","content":[{"type":"input_text","text":"What is in this image?"},{"type":"input_image","image_url":img_url}]}] [S2].
• Vision (base64 data URL):
• image_url=f"data:image/png;base64,{b64_image}" in an input_image part [S2].
• Streaming (sync):
• stream = client.responses.create(model="gpt-5.2", input="Write a one-sentence story.", stream=True); for event in stream: print(event) [S2].
• Useful response field:
• response.output_text holds the generated text in non-streaming mode [S2].

Answers to research questions (evidence-limited to provided sources)

1) Exact request/response schemas, content part types, output structures:

- Request via SDK includes model, input (string or structured with role and content), optional instructions; content parts include input_text and input_image with image_url supporting HTTP or base64 data URL; stream=True enables SSE [S2].
- Response exposes output_text; full schema and additional fields are not documented in the provided source [S2].

2) Supported models and their modality/reasoning/token limits:

- Not documented in the provided source; examples use a model string such as "gpt-5.2" without further details [S2].

3) Structured output via response_format json_schema and strict mode:

- Not documented in the provided source [S2].

4) Streaming mechanics and event types (SSE), tool-call/text assembly:

- Streaming enabled with stream=True and iteration over events; specific event types and assembly details are not documented for Responses in the provided source [S2].

5) Multi-turn state patterns:

- Not documented in the provided source [S2].

6) Tool/function calling differences vs legacy function calling:

- Not documented for Responses; the Realtime API supports function calling over WebSocket but is a different API surface [S2].

7) Error codes, retries, idempotency:

- Not documented for Responses; Realtime sends error events and keeps the connection open without raising exceptions in the SDK [S2].

8) Rate limits, pricing, token usage reporting:

- Not documented in the provided source [S2].

9) Audio/image encoding and constraints:

- Image inputs can be provided via HTTP URL or base64 data URLs within input_image parts; audio handling for Responses is not documented in the provided source [S2].

10) SDK features for responses.create (Python/JS), streaming helpers, typed tool-calling:

- Python SDK supports sync/async clients, streaming (stream=True), vision inputs, and typed requests/responses (TypedDict/Pydantic); aiohttp can be used as an async HTTP backend; typed tool-calling for Responses is not documented in the provided source; JavaScript helpers are not covered in the provided source [S2].

References

- [S1] -> https://platform.openai.com/docs/api-reference/responses
- [S2] -> https://github.com/openai/openai-python