ChatGPT

OpenAI Platform guidance (2026-01-27)

This report summarizes current guidance for key OpenAI platform APIs and tools, including the **Responses API**, **Structured Outputs**, **Batch API**, **Embeddings**, **Vector Stores**, **Images & Vision**, and **Function Calling**. It also describes recommended usage patterns (extraction, reranking and nightly batch processing) and notes pricing and rate-limit considerations. All information comes from official OpenAI documentation.

Responses API

The Responses API is the most capable interface for calling OpenAI language models. It accepts text and image inputs and supports **text**, **image** or **JSON** outputs. Responses can include calls to built-in tools (web search, file search, code interpreter, or custom functions) and can be stateful across conversation messages [1]. You create a conversation by providing a sequence of messages; the system returns the model's response and any tool calls. Parameters include:

- **Tools and multimodal inputs** – messages may include text and images, and the model can decide to call built-in tools. When a tool call is returned you must execute the function and send the result back as a subsequent message [2].
- **Conversation state** – you may supply a `conversation_id` so the API retains previous messages and tool results. Set `store_conversation` to `false` if you do not wish to persist the conversation [3].
- **Background mode** – setting `background` to `true` returns immediately while the model response is generated asynchronously [3].

Pricing & rate limits. The Responses API itself has no special price; tokens consumed for prompts, completions and tool output are billed at the selected model's rates. Built-in tools incur additional costs: file-search storage costs about **$0.10/GB-day**, file-search tool calls cost **$2.50 per 1 000 calls**, web-search calls cost **$10 per 1 000 calls plus input token charges**, and code interpreter sessions cost **$0.03 per session** [4]. Rate limits are measured in **requests per minute/day**, **tokens per minute/day** and **images per minute**; pending batch jobs count against token budgets until completion [5]. Limits vary by model and by organization/project; usage tiers increase monthly allowances as spending rises [6].

Recommended usage

- **Interactive assistants** – use the Responses API when you need the model to choose whether to call tools. Structure conversations with system and user messages and handle tool calls by executing the function and supplying the result.
- **Multimodal interactions** – use image inputs with the `image_detail` parameter to control cost vs. fidelity. Low detail uses a 512-pixel representation and ~85 image tokens; high detail uses higher resolution at greater cost [7].
- **Asynchronous operations** – for tasks that may take time, set `background` to `true` so the request returns immediately and use the returned status endpoint to poll for completion [3].

Structured Outputs

The **structured outputs** feature guarantees that model responses conform to a JSON schema you define, providing reliable type safety. OpenAI notes that structured outputs offer (1) **reliable type-safety** – you control fields and types; (2) **explicit refusals** when content cannot be produced; and (3) **simpler prompting**, since you ask the model to produce objects rather than describing text formatting [8]. Developers can specify the schema using Pydantic models (Python) or Zod (JavaScript/TypeScript); the library validates responses against the schema and throws errors if invalid [9].

A common pattern is **data extraction**: define a schema with fields such as `title`, `authors`, `abstract`, and `keywords` and ask the model to extract these from a paper; the structured output ensures each field is populated correctly [10]. Structured outputs can also be used for summarization, classification, information extraction, or generating well-formed API payloads.

Recommended usage

- **Extraction tasks** – when extracting structured information from unstructured text (e.g., invoices, research papers, news articles), define a schema capturing the desired fields. Use the `structured_output` parameter in the Responses API or assistants to enforce the schema.
- **Downstream automation** – convert untrusted user input into validated JSON for use in databases or APIs. The model either produces a conforming object or a refusal with a `declined` field, enabling safe handling [8].

Batch API

The **Batch API** runs large sets of requests asynchronously. You upload a `.jsonl` file (up to **200 MB** and **50 000 requests**) describing requests to the `completions` or `chat/completions` endpoints and specify a `completion_window` of `24h` [11]. OpenAI processes the batch in the background and stores the results for retrieval. Completions from a batch have a **50 % discount** on input and output tokens [12]. Pending batch jobs count against your token budget until finished [5].

Recommended usage

- **Nightly batch processing** – run periodic jobs that embed documents, classify content, or summarise data. Package requests in a `.jsonl` file, upload to the Batch API, and retrieve results later. This is often cheaper and more scalable than making individual calls because you benefit from the 50 % discount and avoid per-request overhead.
- **Non-interactive workloads** – use the Batch API for large-scale tasks that do not need immediate results, such as re-embedding a corpus after updating your vector store or rerunning analysis on new data.

Embeddings API

Embedding models convert text into high-dimensional vectors that capture semantic meaning. OpenAI's latest embedding models, `text-embedding-3-small` and `text-embedding-3-large`, offer **lower cost**,

better multilingual performance and a parameter to control the output vector length [13]. Embeddings support these limits:

- **Token limits** – each input text may contain at most **8 192 tokens**, and the total tokens across all inputs in a single request cannot exceed **300 000 tokens** [14]. Inputs longer than the model's context are truncated. You can control the dimensionality of the output vector with the `dimensions` parameter (default uses the model's maximum) [15].
- **Use cases** – embeddings are used for **search**, **clustering**, **recommendations**, **anomaly detection**, **diversity measurement**, and **classification** [16]. Similarity is measured using distance functions; small distances mean high relatedness, large distances low relatedness [16].
- **Billing** – you are billed on the number of tokens in the input; there is no cost for output tokens [17].

Reranking and cross-encoders

A common retrieval pattern uses embeddings to obtain candidate documents, then uses a more expensive **cross-encoder** to rerank them. The OpenAI cookbook explains that embeddings (bi-encoders) are inexpensive but may mis-order results; applying a cross-encoder over the top `k` candidates yields higher accuracy. Because cross-encoders process both the query and candidate together, they are more accurate but costly, so they should only be used on a small set of candidates [18]. Total cost is the embedding cost plus the cross-encoder cost per candidate [18].

Recommended usage

- **Search & retrieval** – embed both your documents and user queries, store document embeddings in a vector store, and compute similarity to find relevant items. Use a cross-encoder reranker when higher relevance is critical.
- **Classification & clustering** – compute embeddings for each item and use clustering algorithms or distance metrics to group similar items or classify them.

Vector Stores and File Search

A **vector store** stores embeddings for documents and enables semantic search. Vector stores power the Retrieval API and the `file_search` tool [19]. When creating a vector store, you may supply parameters such as **file IDs**, **chunking strategy** (how to split text into chunks), **metadata**, **name** and **description** [20]. Files are parsed, chunked and embedded; search then retrieves the most relevant chunks. In the assistant workflows, the file search tool automatically parses and chunks documents, creates embeddings, stores them in a vector store and retrieves content using vector and keyword search [21].

Pricing & storage. File search storage is billed at **$0.10 per GB-day** (first GB free) and file search tool calls cost **$2.50 per 1 000 calls** [4]. Content tokens retrieved via file search are billed at the model's input token rate [22].

Recommended usage

- **Document retrieval** – maintain a vector store for your knowledge base. When the user asks a question, embed the query, retrieve the most similar document chunks and optionally apply cross-encoder reranking to improve relevance.

- **Hybrid search** – combine semantic search with keyword filters or metadata (e.g., filter by document type or date). Pass these filters to the vector store to narrow the candidate set.
- **Refreshing embeddings** – if you update your documents or model, re-embed your corpus using the Batch API and update the vector store to keep search results accurate.

Images & Vision

The vision models accept images alongside text and allow the model to describe and reason about visual content. The `image_detail` parameter controls the resolution and cost: `low` uses a **512×512-pixel** representation which consumes about **85 image tokens** per image; `high` uses higher-resolution patches and yields more detailed understanding at greater cost [7]. The documentation warns that the models are **not suited** for medical imaging, small non-Latin text, graphs, rotated images, or tasks requiring precise spatial reasoning [23]. Image input tokens are calculated using a patch-based formula and capped at **1 536 tokens**; each image patch is 32×32 pixels [24].

Pricing. Images are billed based on **image tokens** at the model's rates (e.g., GPT-4 Vision costs $8 per 1M image tokens input and $32 per 1M image tokens output) and image outputs from `image-generators` cost roughly **$0.01–$0.17** per image depending on size and quality [25].

Recommended usage

- **Multimodal QA and summarization** – provide images with your prompts to let the model interpret charts, diagrams or photos. Use `low` detail for simple questions and `high` for complex interpretation [7].
- **Avoid unsuitable tasks** – do not rely on vision models for medical images, fine-grained text recognition, graphs, rotated images or tasks requiring geometric accuracy [23].

Function Calling and Tool Calling

Function calling (also referred to as **tool calling**) allows the model to call user-defined functions or built-in tools. You register a tool by providing its name and parameters as a JSON schema. During the conversation, the model may decide to call a tool; the API then returns a `tool_call` object specifying the function and arguments [26]. You must execute the function with the given arguments and send the result back to the model; this result is called the **tool call output** [26]. This mechanism enables safe integration with external systems (e.g., retrieving weather, querying databases) and ensures the model uses functions only when appropriate.

Recommended usage

- **Clear schemas** – define concise schemas for your functions; include parameter types and descriptions so the model knows when to call them.
- **Handle tool calls** – when the response includes a `tool_call`, run the function with the provided arguments and supply the output as a message in the conversation. Continue the conversation to let the model complete the answer.
- **Built-in tools** – use file search, web search or code interpreter when the model needs external knowledge or code execution. Remember to handle tool costs and rate limits [4].

Rate Limits & Pricing Summary

OpenAI enforces rate limits measured in **requests per minute/day**, **tokens per minute/day**, **images per minute**, etc.; tokens from pending batch jobs count against limits until completion [5]. Limits are set at the **organization** and **project** levels and vary by model; there is also a separate limit for long-context models [27]. OpenAI uses usage tiers—Free, Tier 1, Tier 2, Tier 3, Tier 4 and Tier 5—where higher monthly spending increases request and token budgets [28]. Pricing for each model is on the OpenAI pricing page; for example, the `text-embedding-3` models cost about $0.10 per million tokens input (small) and $0.40 per million tokens input (large), while GPT-4 Turbo costs $10 per million prompt tokens and $30 per million completion tokens (prices may change). Always consult the official pricing page for up-to-date rates.

Summary of usage patterns

Use case	Recommended approach	Key notes
Structured extraction	Define a JSON schema using Pydantic/Zod and call the model with structured outputs; parse responses with `responses.parse`.	Structured outputs ensure type-safety and explicit refusals [8].
Reranking search results	Use embeddings to retrieve candidate documents; apply a cross-encoder or GPT model to rerank top k items.	Embeddings are cheap but approximate; cross-encoders are expensive but accurate [18].
Nightly batch processing	Compile a JSONL file of requests and submit to the Batch API with a 24-hour completion window.	Batches allow up to 50 000 requests and 200 MB files and offer a 50 % discount [29].
Document search with vector stores	Use vector stores or file search to store and retrieve embeddings; filter by metadata; optionally rerank with cross-encoders.	File search storage costs $0.10/GB-day, search calls $2.50 per 1 000; tokens are billed at model rates [4].
Multimodal queries	Provide images with text in the Responses API; use `image_detail` parameter.	`low` detail uses ~85 tokens at 512×512, `high` uses higher resolution [7]; avoid tasks outside capabilities [23].
Function/tool calling	Register functions with JSON schemas; handle tool calls by executing functions and supplying outputs.	Tools enable controlled access to external systems and built-in capabilities [26].

Conclusion

The OpenAI platform provides a cohesive set of APIs and tools for building advanced language-enabled systems. The **Responses API** unifies multimodal prompts, conversation history and tool calling. **Structured outputs** ensure reliable data extraction. The **Batch API** makes large offline jobs affordable. **Embeddings** and **vector stores** unlock semantic search and clustering, while **images & vision** enable multimodal understanding. **Function calling** lets models interact with external services. When designing systems, consider cost and rate-limit constraints and choose the appropriate pattern—real-time responses for interactive applications, batch processing for large offline workloads, and cross-encoder reranking for high-precision retrieval.

---

[1] [2] [3] Responses | OpenAI API Reference
https://platform.openai.com/docs/api-reference/responses/input-items

[4] [22] [25] Pricing | OpenAI
https://openai.com/api/pricing/

[5] [6] [27] [28] Rate limits | OpenAI API
https://platform.openai.com/docs/guides/rate-limits

[7] [23] [24] Images and vision | OpenAI API
https://platform.openai.com/docs/guides/images-vision

[8] [9] [10] Structured model outputs | OpenAI API
https://platform.openai.com/docs/guides/structured-outputs

[11] [12] [29] Batch | OpenAI API Reference
https://platform.openai.com/docs/api-reference/batch

[13] [16] [17] Vector embeddings | OpenAI API
https://platform.openai.com/docs/guides/embeddings

[14] [15] Embeddings | OpenAI API Reference
https://platform.openai.com/docs/api-reference/embeddings

[18] Search reranking with cross-encoders
https://cookbook.openai.com/examples/search_reranking_with_cross-encoders

[19] [20] Vector stores | OpenAI API Reference
https://platform.openai.com/docs/api-reference/vector-stores

[21] Assistants File Search | OpenAI API
https://platform.openai.com/docs/assistants/tools/file-search

[26] Function calling | OpenAI API
https://platform.openai.com/docs/guides/function-calling