ChatGPT

Codex App-Server JSON-RPC Request Schema (Upstream)

When integrating with the Codex app-server, the client uses JSON-RPC 2.0 calls to initialize the session and send user inputs. The key request methods are `initialize`, `sendUserTurn`, and `sendUserMessage`, each with specific JSON payload schemas. Below we outline each request type, including how to incorporate tools (function calls), parallel tool usage, generation parameters (e.g. temperature), and metadata considerations for a complete integration.

Initialize (Handshake)

The `initialize` request is sent once when the app-server starts, to establish the connection and exchange capabilities. Its parameters include a `client_info` object identifying your client application [1]. For example:

- **client_info.name** – Your client's name or identifier (string).
- **client_info.title** – *Optional* human-readable title (string) for the client.
- **client_info.version** – Your client's version (string) [1].

Example: A JSON-RPC request might look like:

```
{
  "id": 1,
  "method": "initialize",
  "params": {
    "client_info": { "name": "MyProxy", "title": "Codex API Proxy", "version":
"1.0.0" }
  }
}
```

The app-server will respond with an **InitializeResponse** (e.g. containing a `user_agent` string) if the handshake succeeds [1]. This step ensures the server is ready before sending any chat requests.

sendUserTurn (Begin User Turn)

Use `sendUserTurn` at the start of each new user turn (conversation round). This request sets up the conversation context for the turn – including model details, tools, and environment – and prepares the server to receive the user's message [2]. The **params** for `sendUserTurn` include the following fields [3]:

- **conversationId** – The unique ID of the conversation (string). If starting a *new* conversation, you can first call `newConversation` to get an ID, or pass a placeholder and let the server create one (the server will return the real ID in a start event) [4]. For subsequent turns in the *same* conversation, reuse the existing ID.
- **items** – An array of input content items (**InputItem**). Each item represents a piece of user input, such as a text message or other data. Typically this will be an InputItem of type `text` carrying the user's prompt (or can be left empty here if the actual message will be sent via `sendUserMessage` next). InputItem can also handle other input types like images if the model supports them [5] [6]. (See **Tools & Function Calls** below for how to include tool outputs as special items.)
- **cwd** – The current working directory (string) for this session. This gives the model context about the project's filesystem location (used if the model tries file operations, etc.) [3].
- **approvalPolicy** – The policy for approving code execution or shell commands (e.g. `"untrusted"`, `"on-request"`, `"on-failure"`, or `"never"`). This controls whether the model can execute commands directly or must ask for user approval [3] [7].
- **sandboxPolicy** – An object defining the sandbox mode for tool execution (e.g. read-only vs. write access). For example, modes include `"read-only"`, `"workspace-write"`, or `"danger-full-access"`, possibly with additional settings (like allowed network access) [3] [8]. This ensures any code the model runs is constrained appropriately.
- **model** – The model name to use for this turn (string). For instance, `"gpt-5-codex"` or another model alias supported by your Codex server [3]. This might be fixed if the app-server was launched with a single model, but can be specified per turn if needed.
- **effort** – *Optional.* Reasoning effort level (enum) for the model's chain-of-thought, if supported. Values might be `"minimal"`, `"low"`, `"medium"`, or `"high"` [3]. Higher effort could mean the model spends more time in self-reflection or generates more reasoning content.
- **summary** – Reasoning summary setting (enum), controlling how the model summarizes its reasoning. Possible values include `"none"`, `"concise"`, or `"detailed"` [3]. This can influence whether the model returns a distilled reasoning overview along with its answer.

When you send `sendUserTurn`, the app-server will acknowledge and start the turn. In the **migration plan**, it's recommended to call `sendUserTurn` first for each incoming API request, which "optionally" sets/receives the `conversation_id` and configures the turn [4]. Typically, you would follow up immediately with a `sendUserMessage` call to provide the user's actual message content.

Note on conversation setup: If you use the separate `newConversation` method instead of `sendUserTurn` to initialize a session, you can provide similar parameters there (model, sandbox, etc.) along with special flags for tools. For example, `NewConversationParams` includes booleans like `include_plan_tool` or `include_apply_patch_tool` to enable those specific tools in the session [9]. These allow the model to use the plan-generation tool or the code patching tool if needed. (When using `sendUserTurn` directly to start a turn, those tools default to off unless the profile/config enables them.)

Generation parameters: Any OpenAI-style generation settings (temperature, max tokens, etc.) are typically passed via the `config` in `NewConversationParams` or via the server's config files. The `config` field accepts a map of settings that override defaults [10]. For example, to adjust randomness or output length for a turn, you could include keys like `"temperature": 0.7` or `"max_tokens": 500` in this config map. (The Codex CLI's flags `--temperature 0.7` and `--max-tokens 500` correspond to these settings [11].) The app-server will apply these to the model's generation for that turn.

sendUserMessage (User Message Content)

After initializing the turn, send the user's actual message text with `sendUserMessage`. This request carries the chat message(s) that the user wrote. Its parameters are simpler:

- **conversationId** – The conversation ID (string) for the ongoing session (must match an existing conversation, e.g. the one started in `sendUserTurn`) [12].
- **items** – An array of InputItem(s) representing the message content [12]. In practice this is usually a single **Text** item containing the user's prompt or question. For example: `items: [ { "type": "Text", "text": "What is the next step?" } ]`. (If the user turn included multiple input parts – e.g. an image plus a question – you would have multiple items here.)

Usage: In the typical flow, you call `sendUserMessage` immediately after `sendUserTurn` to deliver the user's query for that turn [4]. The migration guide instructs that for each API call, you should do `sendUserTurn` then `sendUserMessage` with the full prompt text [4]. The prompt text may be a concatenation of prior conversation context if you are implementing a stateless API – in other words, if the external API provided a list of messages, you might join them into one combined prompt string for this call (since the Codex conversation is fresh per request). The `sendUserMessage` triggers the model to begin processing the user's input and generate a response.

Once `sendUserMessage` is sent, the model will stream its response as events (which you'll receive as JSON notifications over the RPC channel, or via SSE if using the HTTP interface). These include `agentMessageDelta` events for partial answer chunks, `agentMessage` for the final answer, and possibly tool invocation events, which we address next [13].

Tools and Function Calls (Handling Tool Usage)

Codex models can call **tools/functions** during a conversation turn (for example, to run a shell command, read a file, do a web search, etc.). Handling tools involves two aspects: *providing tool outputs back to the model*, and *enabling or hinting parallel tool usage*.

Built-in Tools: The Codex app-server has a set of built-in tools (often called functions) that the model may invoke if enabled – such as executing local shell commands, applying code patches, or performing searches. You control tool availability mainly when starting the conversation/turn (as noted, via parameters like `include_apply_patch_tool` or through the model profile) [9]. If a tool is not enabled, the model will refrain from using it. There is no need to manually specify a list of tools in each request; the model *knows* about its tools and will decide when to use them. (Unlike the OpenAI Chat API's dynamic function definitions, here tools are pre-defined by the system or model.)

Model calling a tool: When the model decides to use a tool, you will receive a sequence of events indicating a function call (for example, `response.output_item.added` with `type:"function_call"`, followed by streaming `response.function_call_arguments.delta` events with JSON arguments, etc.). Eventually a `response.output_item.done` will mark the end of that tool invocation. At that point, the model **pauses** waiting for the tool's result. You must execute the requested tool action on your side (e.g. run the command or query) and then **send the result back to the model** on the next turn before it can continue [14] [15].

To return a tool's result, you will start a new turn (another `sendUserTurn`/`sendUserMessage` sequence) with an **input item of type** `function_call_output` containing the call's outcome [16]. In practice, this means the `items` array in your next `sendUserTurn` should include an object like:

```
{ "type": "function_call_output", "call_id": "<the call ID>", "output": "<tool
output JSON/string>" }
```

This special item tells the model the function's return value, so it can incorporate it into its reasoning. The `call_id` should match the ID provided by the model for that function call. For example, if the model requested `call_7` and you got `tool_call_started` with `call_id":"call_7"`, you would respond with `{"type":"function_call_output","call_id":"call_7","output":"...result..."}` [16]. (If the tool execution failed or returned an error, you can still provide an output and possibly set a flag in the content indicating failure; the schema also allows an optional `success: false` in the output payload in some versions.)

After sending the `function_call_output` item turn, the model will resume its answer, typically producing an assistant message next [14] [15]. Your proxy should continue streaming events to the user as that response comes in.

Parallel tool calls: Newer Codex models (e.g. GPT-4/5 based agents) may request **multiple tools in parallel** to save time. In such cases, the model will emit several function call items in succession without waiting for the first to complete (e.g. `function_call` item for `call_7`, then another `function_call` item for `call_8`, etc., each with their own arguments). There is **no extra parameter needed** to enable this – the agent decides autonomously when to call tools in parallel [17] [18]. Your integration just needs to handle it: execute all the requested tools concurrently (since they are independent) and then return **multiple** `function_call_output` results together in the next turn. In other words, if the model made *n* parallel tool calls, your next `sendUserTurn` should include *n* `function_call_output` items (one for each `call_id`). The model will then use all those results and continue its response. This parallel execution can greatly speed up complex tasks (for example, reading multiple files at once) [19] [17].

Tip: Always preserve the order of tool outputs if the order matters, and match each `call_id` exactly. The JSON-RPC interface and SSE adapter are designed to handle these correctly, emitting the tool results in the sequence they are added.

Metadata and Other Response Fields

The Codex Responses API (downstream) can include a `metadata` object in the final response data. This might contain system or model metadata (e.g. trace info, model identifiers, or other internal details). According to the migration guidelines, your proxy should **sanitize or filter the metadata** before passing the response to the end-user [20]. In practice, this means allowing only whitelisted keys or expected values in the `metadata` and stripping any large or sensitive blobs. The goal is to preserve useful information (if any) while ensuring no internal-only data leaks out.

Additionally, token usage info is provided either in the `response.completed` event or via an explicit include option. If you need usage metrics, set the request's `stream_options.include_usage` to true (when using the HTTP `/v1/responses` endpoint or equivalent) so that the final event contains a `usage` field with token counts [21] [20]. This `usage` data can be treated as part of the metadata/response and forwarded to the user as needed.

Summary: In summary, to unblock your development: use `initialize` once per session to handshake, then for each user query do a `sendUserTurn` (with the appropriate conversation id, model, policies, etc.) followed by a `sendUserMessage` carrying the user's message [2]. Encode tool interactions by reading the model's function call events and replying with `function_call_output` items containing results [16]. You do not need special flags for parallel calls – the model will handle that logic, and you just return all outputs together [18]. Ensure that any generation parameters (like temperature) are passed via the config or conversation setup, and be mindful of filtering the response metadata before returning the final answer to the user [20]. With the above request schemas and examples, you should be able to construct the correct JSON-RPC envelopes in `src/handlers/chat/request.js` and continue your integration. Good luck!

Sources: The schemas and usage were drawn from the Codex app-server protocol definitions and migration guides [1] [3] [12] [18] [20], which provide the authoritative reference for these request formats and behaviors.

---

[1] [5] [6] [9] [10] protocol.rs
https://github.com/0xSero/codex-local/blob/629c4fa7b74d8deae2cdedadee61ebaa1c447c57/codex-rs/app-server-protocol/src/protocol.rs

[2] [4] [13] codex-completions-api-migration.md
https://github.com/DrJLabs/codex-completions-api/blob/4220f15a139c078d37961dd0bf8cca91f55d7d34/docs/app-server-migration/codex-completions-api-migration.md

[3] [7] [8] [12] types.py
https://github.com/gersmann/codex-python/blob/fbb83ce24d2a1b4f4e3eb96126a03a6307784dba/codex/protocol/types.py

[11] Truefoundry Docs
https://docs.truefoundry.com/gateway/openai-codex-cli

[14] [15] [16] [20] [21] codex-app-server-streaming-contract.md
file://file_00000000474071f78c97c4332fa40774

17 18 19 Parallel Tool Calls - Augment Code
https://www.augmentcode.com/changelog/parallel-tool-calls