ChatGPT

Migrating from `codex proto` to `codex app-server` – Input/Output Differences

Background: Changes in Codex CLI Entry Points

Recent updates to OpenAI's Codex CLI have deprecated the old `codex proto` command in favor of a new `codex app-server` mode [1] [2]. In version 0.44.0 and beyond, the `codex app-server` is now the supported entry point for external integrations, replacing the prior "protocol" mode. This means that projects which previously spawned Codex via `codex proto` (a JSON-over-STDIO protocol stream) must transition to using `codex app-server`. Below, we explain the key differences in how you **launch Codex**, **send input**, and **receive output** between the two modes, and what you'll need to change in your integration.

Launching the Codex Process

`codex proto`**:** This command launched Codex in a headless "protocol stream" mode that communicated via STDIN/STDOUT. You would typically run `codex proto` with any needed config flags (for example, specifying a model via `-c 'model="..."'`) and then interact with it by writing JSON lines to its STDIN and reading JSON lines from STDOUT [3]. There was no network service – the integration was purely through pipes. No special handshake was required on startup; the client could begin sending commands (like a user prompt) immediately.

`codex app-server`**:** This command launches Codex as a local **application server** [4]. Despite the name "server," it also uses a streaming JSON-over-STDIO interface by default (primarily for development/debugging use). In other words, you will still run it as a subprocess in your Docker container and communicate via its standard input/output streams. However, the protocol it speaks is more formal: it implements a JSON-RPC 2.0–style API (with some slight tweaks) for bidirectional communication. Notably, the **JSON-RPC header** is omitted in each message for brevity, but the message format (with `id`, `method`, `params`, etc.) follows JSON-RPC conventions [5]. There is **no separate** `proto` **binary** anymore – you simply start the app-server with `codex app-server` (plus any `-c` overrides as needed) [6].

In summary: To start Codex in the new mode, use `codex app-server` instead of `codex proto`. The Codex CLI no longer supports `proto` as of the newer version (the command was removed entirely [1]). Ensure your Docker container or startup scripts invoke the new subcommand.

Input Differences (Client → Codex)

Once the Codex process is running, the way you send commands/prompts to it has changed significantly:

- **Initial Handshake:** Under `codex proto`, there was essentially no formal handshake; you would typically immediately send a JSON message to begin a conversation (more on that below). In `codex app-server`, the first thing your client **must do** is send an **initialize request**. The app-server expects a JSON-RPC request to the method `"initialize"` containing a **ClientInfo** (e.g. your integration's name/version) [7]. For example, your client might write something like:

```
{ "id": 1, "method": "initialize", "params": { "client_info": { "name":
"MyProxy", "version": "1.0" } } }
```

(The exact fields for `client_info` are name, version, and an optional title.) If the app-server is not initialized first, it will reject other requests with an error – it enforces a one-time handshake [8]. Upon receiving this, Codex replies with an **InitializeResponse** (containing a user-agent string) and marks the connection as initialized [9]. This handshake step did **not exist** in proto mode, so it's a new required step in your integration.

- **Sending a User Prompt:** In **proto mode**, the client had to manually orchestrate the start of a conversation turn. Typically, you would send a JSON line to **open a new user turn**, then another with the actual user input text. For example, one would send an object like `{"id": "...", "op": {"type":"user_turn"}}` followed by `{"id": "...", "op": {"type":"user_input","items":[{"type":"text","text":"<YOUR PROMPT>"}]}}`. The first message signaled the agent to start listening for a user message, and the second carried the message content [3] [10]. This two-step sequence was required for each interaction. (The `codex proto` help string described it as "Run the Protocol stream via stdin/stdout," but offered little guidance beyond that [3], hence needing to reverse-engineer the correct sequence.)

Under `codex app-server`, the interaction is framed as JSON-RPC **requests** rather than ad-hoc ops. The protocol defines methods for sending user messages/turns – notably, methods called `sendUserTurn` and `sendUserMessage` (as seen in the app-server schema) [11]. In practice, this still corresponds to a two-step process, but it's encapsulated in named RPC calls: - First, you likely call `sendUserTurn` to begin a new turn (optionally specifying which conversation/thread it belongs to, if not the default). This is analogous to the old `"user_turn"` op. It may return an acknowledgment (e.g. a Turn ID or an empty success response). - Next, call `sendUserMessage` with your message content (the prompt) as a parameter. This carries the actual text of the user query (and any attachments or code snippets, if those are supported as structured items). This is analogous to the old `"user_input"` op.

After these calls, Codex will start processing the prompt. (It's possible the API might allow combining these if a conversation is already active, but the presence of separate turn and message calls suggests you should invoke both in sequence for a new prompt, mirroring what proto did.)

- **Conversation Management:** One major improvement in app-server mode is explicit support for managing conversations (threads) via the API. The old proto mode implicitly worked with a single session unless you restarted the process. In the new API, there are methods like

`newConversation` (to start a fresh conversation/thread) and `resumeConversation` (to switch context to a saved thread) [12] [13]. In many simple use-cases you may not need to call these explicitly – the app-server might auto-create a default conversation when you start a turn. Indeed, upon startup or first use, Codex app-server will configure a session (thread) and inform the client (see output differences below). However, if your proxy wants to isolate each request as a separate session (to avoid any cross-talk in memory), you should utilize `newConversation` for each new independent query. The API returns a `conversation_id` (thread ID) which you then reference in subsequent calls (the **Session/Thread ID** can also be seen in the notifications Codex sends). This explicit control was not available with `codex proto` – in proto you'd have to restart Codex to truly get a clean session. With app-server, you can manage multiple threads within one process if needed.

Summary of Input Changes: In code, you'll replace writing `{"op":"user_turn"}` / `{"op":"user_input": ...}` with sending JSON-RPC formatted requests like `{"method":"initialize",...}`, then `{"method":"sendUserTurn",...}` and `{"method":"sendUserMessage",...}`. Every request should include a unique `id` (for matching responses). Also be mindful of conversation IDs if you manage multiple threads. Essentially, **the interaction is now a structured request/response protocol** rather than an open stream of ops – you initiate actions by method name, and the CLI confirms or responds to each, while streaming events back.

Output Differences (Codex → Client)

The switch to JSON-RPC in `codex app-server` also changes the format and content of the output you receive from Codex. Here's what to expect:

- **Structured Responses vs. Raw Events:** In proto mode, Codex's STDOUT emitted a series of JSON "event" objects as things happened. For example, right after sending a prompt you would typically get a `session_configured` event (indicating the session and model info) followed by a stream of events representing the AI's actions and messages (tool uses, code execution results, etc.), culminating in an `assistant_message` event which contained the final answer text from the AI. These were often delivered under a generic `"op"` or `"msg"` field with a type indicator. There was no notion of matching responses to requests – it was a continuous event stream.

In app-server mode, the output is framed in terms of **JSON-RPC responses and notifications**: - When you send a request, if it completes quickly or has a return value, you'll get a JSON-RPC **response** with the same `id`. For instance, the initial `"initialize"` request returns a response containing a `user_agent` string (to confirm login info) [9]. Many control requests (like `newConversation`) will likewise return a response (often with an ID for the new resource, etc.). - However, much of what the Codex agent does is asynchronous and streaming (the AI "thinking" and producing output). These come to the client as JSON-RPC **notifications** (which have a `"method"` and `"params"`, but no `id`). The app-server will **push events** to you using specific method names for each event type. In other words, instead of an `"op":"assistant_message"` event, you might receive a **notification** like:

```
{"method": "agentMessage", "params": { ... message content ... }}
```

This would represent the AI agent producing an answer (more on naming changes below). Because it's a notification, it's not a direct reply to a particular request – it's part of the streamed result of the earlier `sendUserMessage` call.

- **Initial Session Setup Event:** One of the first notifications you'll receive (after initialization) is an event indicating the conversation/session has been set up. In proto, this was an event like `codex/event/session_configured` carrying info such as the session ID, selected model, and history context. In app-server, there is an analogous notification, now typically called `sessionConfigured` (or `threadStarted` after recent terminology updates – see below). Codex will emit a notification when a new conversation is created or resumed, including fields like the `session_id` (thread ID), the model in use, any initial system messages, etc. [14] [15]. Your integration should listen for this, especially if you need the conversation ID or confirmation that the agent is ready. (If you explicitly called `newConversation`, the response to that might also contain the ID, but the notification provides the full context.)

- **Streaming of the Assistant's Answer:** As the AI formulates a reply, Codex may stream the content in parts. With proto, the final answer might arrive as one `assistant_message` event (possibly containing the whole text or broken into chunks in an array of `items`). In app-server, you will likely get one or more `agentMessage` notifications for the assistant's output. Each such notification contains a segment of the assistant's message. For example, you might receive:

  ```
  {"method": "agentMessage", "params": {"text": "<some portion of answer>"}}
  ```

  followed by additional `agentMessage` notifications, and finally perhaps a completion notification (like a `turnComplete` or similar). The exact chunking behavior depends on how the CLI streams content, but be prepared for multiple notifications. Only once the assistant's answer is fully delivered will the app-server consider the turn complete. In some cases, the *final* piece of the answer or the end-of-turn signal might come as a JSON-RPC **response** to your `sendUserMessage` request (since the request can be considered fulfilled when the assistant has answered). The Codex app-server protocol defines a `SendUserMessageResponse` which could carry a summary or final status [11] – for instance, it might indicate that the turn finished successfully. Keep an eye on the responses to your requests in addition to notifications.

- **Event Name and Field Changes:** Many of the JSON keys and event names have changed between proto and app-server:

- **"Assistant" is now "Agent":** The term *assistant* has been replaced with *agent* in the naming. A notable example is the **assistant_message** event type – in the new protocol this is renamed to **agent message**. A recent change explicitly *"Rename[d] assistant message to agent message and fix[ed] item type field naming"* [16]. Practically, this means where your old code might have looked for `event.op.type == "assistant_message"` or similar, you now should expect something like `"method": "agentMessage"` in notifications. Likewise, any JSON structure that included `"type": "text"` for message segments might use a slightly different schema now (they adjusted some field names for consistency).

- **Sessions are now Threads:** The concept of a conversation session has been relabeled as a *thread* in parts of the CLI. For example, what was internally called a session ID might now be referred to as a thread ID. In the changelog, they mention *"rename session created to thread started"* [17]. So you may see notifications or responses using the term *thread* for conversation identifiers. Functionally it's the same idea – just nomenclature to be aware of when reading events or using API calls (`listConversations` might list threads, etc.).

- **JSON-RPC envelope:** Each message coming from Codex in app-server mode will indicate its nature by presence or absence of certain fields:

  - **Notifications** (pushed events) have `"method"` and `"params"` fields, and no `"id"`.
  - **Responses** (replies to your requests) have an `"id"` matching the request and either a `"result"` or `"error"` field. For example, after you send `initialize`, you get a response with `"id":1` and a `"result": {"user_agent": "codex/..."}"`.
  - In proto mode, messages did not follow this structure; they often had a top-level `"id"` (which was more of an event ID, not tied to requests) and some nested object indicating the event type. Your parsing code will need to switch to checking for `method` names, etc., instead of the old format.

- **Tool and Execution Events:** If your agent uses tools or executes code (since Codex can run shell commands, etc.), those events will also come as structured notifications. For instance, there might be notifications like `"toolStarted"`, `"toolOutput"`, `"toolFinished"`, or execution approvals. These correspond to things you might have seen in proto as events when the agent ran a command or required approval. The app-server protocol formalizes these with distinct methods and payloads. Be prepared to handle or ignore them as needed. (For example, if your proxy just wants the final answer and to pass it along, you might ignore tool-related notifications; but if you want to log them or enforce sandboxing, you could watch for them.)

- **End-of-Turn / Completion:** In proto, it might not have been obvious when the agent finished its answer except by the absence of new events or perhaps a special event indicating the agent is awaiting user input again. In app-server, because the interactions are request-based, you have a clearer signal: the `sendUserTurn` / `sendUserMessage` cycle will eventually yield a final response or a notification that the turn is done. There is mention of a `turn.failed` event in the changelog (to notify if something went wrong in processing) [17], and presumably a successful completion would either implicitly be known when you get the `SendUserMessageResponse` or via a `turnCompleted` notification. The exact method name for successful turn completion isn't explicitly noted in our sources, but your integration should watch for either the final response to your message request or a notification indicating completion, to know when to stop reading output for that query.

Summary of Output Changes: The Codex app-server will send more structured and differently named messages. Your proxy code needs to shift from handling a raw stream of `op/type` events to handling **JSON-RPC notifications and responses**. In practice, this means checking for `"method"` names in incoming JSON. Expect a `sessionConfigured` (or `threadStarted`) notification early on, then one or more `agentMessage` notifications carrying the assistant's answer, and possibly a closing indication. All of these come asynchronously while the original request is being processed. Also adjust for naming differences (agent vs assistant, etc.) when extracting the content. For instance, if previously you grabbed

`event["op"]["items"][0]["text"]` for the answer, now you might grab `notification["params"]["text"]` (depending on the exact schema of agentMessage).

Other Considerations

- **Feature Support:** The new app-server mode is under active development, so some features that worked with proto might behave slightly differently. For example, community reports note that Codex 0.44.0's app-server currently **only fully supports OpenAI's own model providers**; custom providers (like Gemini or local/Ollama backends that some users wired up in proto mode via `model_provider=oss`) may not yet work with app-server [18]. This is likely due to Codex now using an internal "responses API" for model queries instead of the old chat-completions approach [19]. If your project only uses the OpenAI models through Codex (which is the default case), this won't impact you. But if you had configured Codex to use a local model in proto mode, be aware it might not plug-and-play in app-server mode without further updates.
- **Performance and Stability:** The **codex app-server** is intended as a more robust integration point. It splits responsibilities more cleanly (there's also a `codex mcp-server` for low-level Model Context Protocol over stdio, but that is more for tool integrations, whereas app-server is for full app/IDE integration). In general, you should find the app-server's JSON-RPC interface easier to work with once set up, as it clearly delineates requests and events. It is also likely to receive more attention going forward, whereas `proto` was undocumented and has now been removed [20] [1].
- **Updating Your Proxy Code:** To adapt your existing proxy:
- **Startup:** Change the command it launches from `codex proto` to `codex app-server`. Ensure any config flags (like model selection, etc.) are still passed via `-c` as needed – those work globally on Codex CLI commands [21] [22].
- **Handshake:** Implement the initialize step. The proxy should read the initialization response before proceeding. (If running in a non-interactive environment, make sure Codex is already authenticated – you might need to handle `codex login` outside of this, or use an API key with `--with-api-key` if appropriate.)
- **Message format:** Refactor how you send the user's prompt. Use the JSON-RPC format (`method` and `params`). At minimum, call `sendUserMessage` (and possibly `sendUserTurn` first). Populate the params with the user's question (and any additional metadata required).
- **Parsing output:** Rewrite your parsing logic to handle JSON-RPC notifications. For example, listen for `"method":"agentMessage"` to gather the assistant's answer text. You may need to accumulate text across multiple such messages. Also handle or ignore other notifications (e.g., tool usage or debug info) as appropriate. When you see the conversation has ended (either via a final response or known notification), you can consider the Codex answer complete and return it through your proxy as the OpenAI API-style response.
- **Testing:** Because the protocol "may change without notice" (as the docs caution [4]), test your integration with the specific Codex CLI version you're targeting. Check the Codex CLI changelog for any recent adjustments to the app-server protocol. For instance, if updating beyond 0.44.0, verify if any field names or method names changed (the team has been tweaking names like session vs thread, etc., as noted above).

By accounting for these input/output differences, you should be able to successfully migrate your project from using `codex proto` to the new `codex app-server` interface. The new mode provides a more standardized and extensible way to interact with Codex, which in the long run should make integrations more robust.

References:

- OpenAI Codex CLI Changelog – entries for removal of `proto` and introduction of `app-server` [1] [2], and naming updates [16].
- GitHub issue demonstrating `codex proto` usage and lack of documentation [3] [10].
- Codex CLI developers' discussion of the JSON-RPC handshake and client update requirements (initialize step) [7] [9].
- Community notes on switching from proto to app-server in Codex 0.44 (Codexia release notes) [18].

---

[1] [2] [16] [17] Codex changelog
https://developers.openai.com/codex/changelog/

[3] [10] [20] Documentation for `codex proto` missing · Issue #3447 · openai/codex · GitHub
https://github.com/openai/codex/issues/3447

[4] Codex CLI reference
https://developers.openai.com/codex/cli/reference/

[5] Codex CLI: codex-rs/app-server/README.md | Fossies
https://fossies.org/linux/codex-rust/codex-rs/app-server/README.md

[6] [21] [22] Codex CLI 完全ガイド：全オプション徹底解説 #AI - Qiita
https://qiita.com/nogataka/items/104f0c906d9a8a4af1d2

[7] [8] [9] message_processor.rs
https://github.com/openai/codex/blob/13e1d0362d4500d78db0e0c7e7b5314ff514ca30/codex-rs/app-server/src/message_processor.rs

[11] [12] [13] export.rs
https://github.com/openai/codex/blob/13e1d0362d4500d78db0e0c7e7b5314ff514ca30/codex-rs/app-server-protocol/src/export.rs

[14] [15] fix: separate `codex mcp` into `codex mcp-server` and `codex app-server` by bolinfest · Pull Request #4471 · openai/codex · GitHub
https://github.com/openai/codex/pull/4471

[18] [19] milisp/codexia v0.7.0 on GitHub
https://newreleases.io/project/github/milisp/codexia/release/v0.7.0