ChatGPT

Replace `codex proto` with `codex app-server` in CLI Invocation

All places that build or use the Codex CLI arguments must be updated to call the **app-server** subcommand instead of the legacy **proto** mode. For example, in `buildProtoArgs` (probably renamed to `buildAppServerArgs`), the first argument should change from `"proto"` to `"app-server"`. Currently it constructs an array like:

```
const args = [
  "proto",
  "--config", 'preferred_auth_method="chatgpt"',
  … // other --config flags
];
```

This must become:

```
const args = [
  "app-server",
  "--config", 'preferred_auth_method="chatgpt"',
  …
];
```

All the configuration overrides (model, sandbox, etc.) can remain – the Codex CLI still accepts them on the app-server command. For instance, we'll still pass the model (`model="${effectiveModel}"`), sandbox mode, `parallel_tool_calls`, reasoning effort, etc., exactly as before [1]. The key difference is that we no longer invoke the one-shot "proto" RPC mode; instead we launch a persistent **Codex app-server** process.

In logs and env vars, any references to "proto" should be revised to "app-server" for clarity. For example, the dev log line in the streaming handler currently says `"[proxy] spawning (proto): …"` [2] – this should say "spawning (app-server)" since we are now starting the app-server. Similarly, config like `PROXY_DEBUG_PROTO` could be renamed (or repurposed) to something like `PROXY_DEBUG_CODEX` or `PROXY_DEBUG_APP_SERVER` to avoid confusion. The `PROXY_PROTO_IDLE_MS` setting (idle timeout for proto processes) will either be removed or replaced, since we won't be spawning per-request processes that automatically exit – more on this below.

Spawning and Lifecycle of the Codex Process

Instead of spawning a new Codex CLI process for each request, we will run a single persistent `codex app-server` subprocess and send it commands for each API request. In the current implementation, every incoming completion request triggers `spawnCodex(args)` to start a fresh child running `codex proto …`, then writes the prompt to its stdin [3], streams out results, and finally kills the

process when done [4] [5]. With `codex app-server`, we should shift to a long-lived backend service model:

- **Launch the app-server once** (e.g. during server startup or on first request) by spawning `codex app-server ...` with the same environment and config flags. This process will stay running, listening for tasks. We might create a utility in `codex-runner.js` to initialize this singleton. For example:

```
// Pseudocode
if (!codexAppServerProcess) {
   codexAppServerProcess = spawnCodex(appServerArgs);
   // set up listeners on stdout, etc.
}
```

  We'll want to log that we started the app-server (similar to the existing console for spawning proto).

- **Do not spawn on each request**. Instead, for each incoming API call, use the existing child process. This means modifying the handlers. For instance, in `postChatStream`, instead of `const child = spawnCodex(args)` for every call [6], we'll obtain the single `codexAppServerProcess` (spawning it if not already started). We must also remove the per-request `child.on('close', …)` cleanup logic that assumed one process per call – the app-server isn't expected to exit after one completion.

- **Graceful shutdown**: ensure that when our Node server shuts down (SIGINT/SIGTERM), we also terminate the Codex app-server process. In `server.js` we already trap signals to close the HTTP server [7]; we should additionally call something like `codexAppServerProcess.kill()` there so the CLI doesn't linger in the background.

JSON-RPC Communication and Message Formatting

The Codex app-server uses a JSON-RPC 2.0 protocol over its stdio, rather than the simple JSONL "event" stream that `codex proto` produced [8]. This requires changes in how we send user prompts and how we parse responses:

- **Submitting Requests:** Under `proto`, the Node code wrote a single JSON object with `{ id, op: { type: "user_input", items: [...] } }` to stdin [3], where the `items` array contained our prompt text. The app-server expects JSON-RPC requests. We will likely need to send a **"create conversation" or "execute prompt" request**. For example, the app-server may support a method like `"conversation.create"` or `"assistant_message"` to initiate a new task. In practice, we might send something like:

```
// Example: create a new conversation with the joined prompt
codexAppServerProcess.stdin.write(JSON.stringify({
  jsonrpc: "2.0",
  id: reqIdNum,
```

```
  method: "conversation/create",
  params: { prompt: promptText /* joined messages */ }
}) + "\n");
```

Here `promptText` is the same string we currently construct (via `joinMessages(messages)` with "[role] content" lines [9]) to preserve the full history context. Alternatively, the protocol might require separate steps: e.g. first a method to create a conversation (getting a conversation ID), then another to send the user's message. In that case, our code would need to perform both calls sequentially. The key point is that **instead of a single JSON** `user_input` **payload, we issue one or more JSON-RPC calls** to feed the conversation to Codex.

We also need to generate a unique RPC **ID** for each request we send (distinct from our `reqId` used for logging) and track it so we can match the response. The `reqId` (used internally for request logging/telemetry) can still be used as a basis, but it's a string like `'nanoid()'`; JSON-RPC ids are often numeric or strings – we can use a numeric counter or hash the reqId string.

- **Streaming Responses:** The app-server will output JSON-RPC **notifications** and responses on stdout. In proto mode, each line was a JSON event with a top-level `"type"` field (e.g. `"agent_message_delta"`) [10] [11]. Now, we should expect messages of the form:
- **Notifications** with a `"method"` and `"params"` (no `id`): these signify events like intermediate tokens or other status. For example, we might get:

  ```
  {"jsonrpc": "2.0", "method": "agent.message.delta", "params": { "delta": "Hel" }}
  ```

  followed by more delta events, similar to the old `agent_message_delta` events. There may also be notifications for things like a tool invocation or a reasoning update. (In the Codex CLI changelog, new event types were added for the app-server, e.g. `codex/event/raw_item` [12].) Our code must be updated to detect these `"method"` notifications and handle them. Concretely, where we currently do:

  ```
  const evt = JSON.parse(line);
  const t = evt.msg?.type || evt.type;
  if (t === "agent_message_delta") { … } else if (t === "agent_message") {
  … }
  ```

  [13] [14], we need to generalize this. For app-server output:

  - If `evt.method` exists, we can derive an equivalent event type. For example, `method: "agent.message.delta"` could be mapped to an internal type `"agent_message_delta"`. We'd then enter the same logic block that appends content to the SSE stream for the client. Similarly, a `method: "agent.message"` (final assistant message) would correspond to the `agent_message` event handling. Essentially, **parse**

JSON-RPC notifications and route them through our existing streaming logic, populating the `delta` text or tool/function call data as before.
    - If `evt.id` and a `"result"` field exist, that's the final RPC response. The result likely contains the completed assistant message (and possibly usage metrics). We should handle this as the end-of-request signal. In streaming mode, we might actually receive all the content via incremental notifications before the final result arrives. We'll probably ignore the final `"result"` in streaming (or use it purely for sanity checks/usage) and rely on the notifications to stream data out. In non-streaming mode, however, we will wait for this final result to build the response.

- **Conversation and Session IDs:** The app-server is session-oriented, so we must manage conversation state:

- **One conversation per API call:** To maintain the same external API behavior (the client must send all prior messages for context), we don't need persistent cross-request memory. We will start a fresh conversation for each `/v1/chat/completions` call. That means after we finish generating a reply, we can consider that conversation done (we may even explicitly close or reset it if the API allows, to free memory).
- If the JSON-RPC interface returns a `conversation_id` (or similar) when starting, we should capture it. We might not actually need to do anything with it if we aren't continuing that convo, but it's good to log it or clean it up. In case the app-server doesn't automatically drop conversations, we might call a cleanup method (like `conversation/delete`) after sending the final answer to avoid accumulating state.
- **Parallel requests:** With a single app-server process, multiple conversations could be in flight. JSON-RPC IDs will help distinguish responses. We'll need to **queue or thread** our requests carefully. If the Codex app-server can handle concurrent calls, we can send them, but we must ensure our SSE streaming for each request uses only the notifications relevant to that request's conversation. This likely means including the conversation ID or request ID in our internal tracking. For example, when a notification comes in, its params might include the conversation or request context. We'll filter or route events to the correct HTTP response. (If concurrency is low, a simpler approach is to serialize access to the Codex process, but ideally we handle up to `PROXY_SSE_MAX_CONCURRENCY` requests in parallel as before.)

In summary, **the code that reads `child.stdout` and parses JSON lines needs a significant update** to handle JSON-RPC structure. We will look for `"method"` vs `"result"` keys instead of `"type"`, and call the appropriate logic. Once adapted, the rest of the streaming pipeline (sending SSE chunks with `delta` content, tool call handling, etc.) can remain mostly the same – the underlying semantics of events like "assistant started typing", "assistant finished message" remain, just wrapped differently.

As a concrete example, a partial sequence might be: the proxy sends a `"conversation/create"` RPC, then Codex emits: - `{"jsonrpc":"2.0","method":"session.created","params":{"session_id":"abc123"}}` (hypothetical) – our code can ignore or log the session start. - `{"jsonrpc":"2.0","method":"agent.message.delta","params":{"delta":"Hello"}}` – our code appends "Hello" to the SSE stream (just as it would for a proto `agent_message_delta` event, sending a chunk with role:assistant and content "Hello"). - More `agent.message.delta` notifications as tokens stream in. - Perhaps a `{"jsonrpc":"2.0","method":"token_count","params":{"prompt_tokens": 42, "completion_tokens": 17}}` – we capture these numbers for the usage

report. - `{"jsonrpc":"2.0","method":"agent.message","params":{"message":{"role":"assistant","content":"Hello world"}}}` – final full message event (or it might come as the RPC result instead). - Finally `{"jsonrpc":"2.0","id":reqIdNum,"result":{"status":"complete"}}` indicating the request finished. At this point we send the SSE terminator `data: [DONE]` and end the response.

The **non-streaming code path** will similarly change. Currently it reads all stdout into a buffer and then parses events to build the final JSON result (with `choices` and `usage`). With app-server, non-streaming can be handled by issuing the RPC call and simply waiting for the final result object. We might not need to parse every notification for content (though we might for assembling function call data or tools). A simpler approach is to accumulate the assistant's message from any delta events, but ultimately the final `result` or last `agent.message` gives the complete content. We then construct the response JSON as before. We will still use our finish reason tracker and metadata sanitizer as appropriate, feeding them with the new event data (e.g. call `finishReasonTracker.record()` on the new finish events).

Session management within a single request: One subtlety is how we feed multi-turn conversations *within* one API call. In the proto implementation, we flattened all prior messages into one prompt string [9]. We can continue doing that for consistency – effectively treating the whole history as one user query to Codex. An alternative, now that Codex can manage a conversation, would be to replay each message via separate RPC calls (e.g. a system message, then each user/assistant pair) so the model truly sees them as distinct turns. However, the Codex app-server's API for injecting assistant messages (e.g. providing examples) is unclear and might not exist. Given that our goal is to maintain identical behavior, **we will keep concatenating the history into a single prompt** and send it in one go. This means Codex still receives a single user prompt like:

```
[system] Your name is ChatGPT
[user] Hi, how are you?
[assistant] I am fine.
[user] Tell me a joke.
```

Codex will then produce the assistant answer to the last user query. This approach ensures the output will be formatted as before, with the assistant possibly including the prior context naturally from the prompt. (If in the future Codex app-server supports natively setting system instructions or example turns, we could refactor to use those, but for now we'll stick to the joined text method to avoid altering model behavior.)

Adjustments to Config and Flags

Beyond code changes, some configuration defaults and flags need updating for the new integration:

- **No backward compatibility mode:** We will remove any toggles or fallback code for `proto`. For example, if there were conditionals like `if (useProtoShim) ... else spawn real Codex`, those can be simplified to always use the app-server. The development shim scripts (`scripts/fake-codex-proto*.js`) will either need to be replaced with an equivalent **fake app-server** script or removed if not needed. (Our tests and dev flows might want a dummy "app-server"

that echoes events, similar to how the proto shim works, so implementing a `fake-codex-app-server.js` that speaks JSON-RPC could be useful for offline testing.)

- **Authentication Config:** The Codex CLI still requires authentication to a model provider (OpenAI) just as before. We are already setting `preferred_auth_method="chatgpt"` in the config args [15], which tells Codex to use your ChatGPT OAuth creds. This should remain. However, **we must ensure the app-server has valid credentials on startup**, since it will likely try to refresh or check auth when launched. In practice, this means the environment's `CODEX_HOME` must contain a prior successful login (or we provide an API key). If we haven't already, we might introduce an option to use API Key auth: e.g. set `preferred_auth_method="api_key"` and ensure `OPENAI_API_KEY` is present in the env. Regardless of method, **the Docker/compose setup should mount or provide credentials** (see below). The phrase "ensure auth with `codex login`" means that before running the proxy, you should run `codex login` (probably using an interactive browser flow for ChatGPT auth or by piping an API key) so that `~/.codex/config.toml` has the necessary tokens. The app-server will read those from `CODEX_HOME`.

- **CODEX_HOME and persistent data:** In `config/index.js`, `CODEX_HOME` default is set to `./.codex-api` in our working directory [16]. We should continue using that. But now, because the Codex process persists, it may write more data to `CODEX_HOME` (conversation archives, logs, etc.). **Mounting this directory** is important so that data (especially auth tokens) isn't lost or locked inside a container. In development, the README already suggests `CODEX_HOME="$(pwd)/.codev"` for local runs [17]. For production Docker, we will update documentation or compose files to mount a volume for `/app/.codex-api`. This allows you to perform `codex login` on the host and then launch the container with the same credentials. If using Kubernetes or similar, one would mount a secret or volume at that path.

- **CLI Flags and Env:** We should verify if any flags need to change for app-server. Many of the `--config` we used with proto are still applicable. One we might *add* is `--json` (or its equivalent) if required to force JSONL outputs. The Codex CLI's release notes indicate that `codex exec` required `--json` for JSON streaming, but `codex app-server` likely always uses JSON-RPC output by design. We can confirm by testing; if not automatically JSON, we'll add `--json` or `--experimental-json`. (Given the CLI docs say app-server is JSON-RPC, we're probably fine without extra flags.)

Another consideration: the **idle timeout**. We had `PROXY_TIMEOUT_MS` (overall request timeout) which we still enforce (we still have a timer that will kill a long-running generation) [18] [19]. But previously we also had a `PROXY_PROTO_IDLE_MS` to forcibly kill a proto child if it sat idle. With a single app-server process, we **should not kill it after each request**; instead, it will be running indefinitely. We might repurpose `PROXY_PROTO_IDLE_MS` as a *session idle* timeout: for example, if the app-server sends no data for that many milliseconds during a generation, perhaps consider it hung. However, our `PROXY_TIMEOUT_MS` already covers total request timeouts, and the app-server itself might not emit an idle event. It's likely we can drop `PROXY_PROTO_IDLE_MS` or set it to 0. We will rely on `PROXY_TIMEOUT_MS` to abort a stuck request (by sending SIGKILL to the process or possibly a JSON-RPC cancel call if supported). We must be careful **not to kill the whole app-server** on one request timeout – ideally we cancel just that task. If Codex app-server doesn't support per-task cancellation via RPC, our fallback is to send a signal (SIGTERM) to terminate the entire process on a fatal timeout, and then restart it fresh (since it's not multi-tenant at that

point). This is an area to document: a long-hanging generation will currently trigger `child.kill("SIGKILL")` [19] – in the new model that could take down the shared backend. We might adjust this to a less drastic measure, or accept restarting Codex and spawning a new one on next request if it hangs.

- **Max concurrency**: No config change here, but note that with a shared backend we should respect `PROXY_SSE_MAX_CONCURRENCY` (e.g. limit to 4 simultaneous streams) by queueing if needed. We already have a guard for SSE concurrency [20]. We'll keep that. If performance or Codex limitations demand it, we might even set the concurrency to 1 (serial requests) in config when using a single process, but if app-server can handle multiple, we try to maintain the same concurrency limit as before.

- **Startup commands and CLI usage:** Running the proxy in development or production might involve slightly different steps now:

- In **development**, previously one could do `npm run start:codev` and that would spawn codex on the fly. This still works, but we should clarify that the proxy now assumes a running codex app-server internally. We might add a convenience script to ensure login. For instance, we could document: *"Before running* `npm start`*, make sure you have logged in with* `codex login` *(your OAuth or API key) so the app-server can authenticate."* The `.env.dev.example` hints that in dev Docker stack, `CODEX_BIN=codex` and one should mount `~/.cargo/bin/codex` (or now the NPM binary) [21]. With app-server, it's the same binary just invoked differently. So no changes to how we specify `CODEX_BIN` (still the `codex` CLI path).
- In **production**, our Dockerfile already installs the Codex CLI (`@openai/codex` npm package v0.49) and sets `CODEX_BIN=/usr/local/lib/codex-cli/bin/codex.js` [22]. We will update that package to the latest version that supports app-server (>=0.47). In fact, since `codex proto` was removed in 0.47 [23] [24], upgrading the CLI is a driving reason for this migration. So we'll bump `@openai/codex` to ^0.50.x in package.json. The Docker build will then bake in the updated CLI.
- The container startup command (`CMD ["node","server.js"]`) remains the same – we're still launching our Node proxy. But now that Node will immediately spawn the codex app-server internally. We need to ensure the container has access to the user's Codex config (via volume) or environment. For example, we might modify our deployment docs to run the container with `-v ~/.codex:/app/.codex-api` (if using the default CODEX_HOME) and perhaps an env var for model if needed. If we want to support non-interactive login via API key in production, we could allow passing an env like `OPENAI_API_KEY` into the container and modify entrypoint: e.g. if that env is present and no existing creds, run `echo "$OPENAI_API_KEY" | codex login --api-key`. This would be a new addition – it ensures headless auth in container. The question specifically mentions "ensuring auth with `codex login`" and mounting CODEX_HOME, so at minimum we document those steps for deployment.

Updating API Surface and Usage Reporting

One requirement is to maintain the existing OpenAI-compatible API output. That means clients should see **no difference in the JSON fields or formatting of responses** (aside from any bug fixes). All changes are

internal. We must double-check that things like the `usage` field and finish reasons are still populated correctly:

- **Finish reasons:** Our finish reason tracker and logic in `finalizeStream` / `finalizeResponse` should still work, but we might receive different raw signals. For instance, previously we relied on the `task_complete` event with a `msg.finish_reason` [25]. In app-server, there might not be a literal `task_complete` event; instead the final RPC result or a conversation closure event might indicate completion. We'll adapt by treating the end-of-response as a "stop" unless we saw an explicit reason. The code already accumulates possible reasons from events (aliases like "length", "content_filter", etc.) [26] [27]. We'll plug in the new events to this mechanism. For example, if app-server issues a notification when it stops due to token limit, we map that to a "length" finish_reason. This way, the `choices[].finish_reason` in the JSON returned to the client remains correct (e.g. "stop", "length", "content_filter", etc., just as before).

- **Tool and Function Calls:** The proxy currently supports tool call and function call outputs by intercepting the `agent_message_delta` and `agent_message` events containing `tool_calls` or `function_call` data [28] [29]. We must ensure the app-server's corresponding notifications are handled so that we continue to populate `choices[].message.tool_calls` or `function_call` in the response. The Codex app-server likely uses the same structure, but perhaps slightly different event naming (for example, it might unify these under the JSON-RPC methods or include them in the final result). We'll test and adjust the parsing. The goal is that a function call or tool usage still appears in the output JSON exactly as it did with proto. (The client shouldn't know that the backend changed.)

- **Usage data:** In the current non-stream flow, after reading all events we compute `prompt_tokens` and `completion_tokens`. We capture them either from the `token_count` event or by fallback estimation [30] [31]. We will continue this. If the app-server returns usage info in its final result (some CLI APIs do include token counts in the response), we can directly use that. Otherwise, we'll listen for a `token_count` notification or similar – likely the app-server still emits a final usage summary event (perhaps named `conversation.tokens`). We saw in the fake proto that a `token_count` event was emitted just before `task_complete` [32] [33]. It's likely the app-server does the same or even includes it in the RPC result. So in our `child.stdout.on('data')` handler for non-stream, we'll add logic:

```
if (evt.method === 'token.count' || evt.method === 'token_count' ||
evt.msg?.prompt_tokens) {
    prompt_tokens = Number(evt.params?.prompt_tokens ??
evt.msg?.prompt_tokens ?? prompt_tokens);
    completion_tokens = Number(evt.params?.completion_tokens ??
evt.msg?.completion_tokens ?? completion_tokens);
}
```

(The exact field might be in `evt.params` now, since JSON-RPC notifications likely put data under `params` instead of `msg`.) By the end of the conversation, we should have real token counts. We then set the `usage` field in the final JSON as:

```
"usage": { "prompt_tokens": X, "completion_tokens": Y, "total_tokens":
X+Y }
```

just as we do now [34]. Maintaining this ensures clients still get usage metrics.

Finally, **the Dockerfile and runtime config** should be reviewed for any needed changes. The Dockerfile currently copies the Codex NPM module and links `codex` in `$PATH` [22], which is fine. We will just update the version of `@openai/codex` in `package.json` to one that supports app-server (0.49 or newer) [35] – in fact, 0.49.0 is already listed, so presumably we're upgrading to 0.50.x. The build process shouldn't need additional steps. At runtime, though, we **must mount CODEX_HOME** as mentioned to provide credentials. For example, if running with Docker Compose, we'd add a volume:

```
services:
  codex-proxy:
    image: drjlabs/codex-completions-api:latest
    volumes:
      - ~/.codex:/app/.codex-api
    environment:
      - CODEX_MODEL=gpt-5
      - PROXY_API_KEY=<your-proxy-key>
      # (and optionally OPENAI_API_KEY if using that auth method)
```

We'll document that the user should run `codex login` on the host (which writes to ~/.codex), or set `OPENAI_API_KEY` and tweak `preferred_auth_method` accordingly.

In summary, the migration involves **replacing the one-request-per-process "proto" integration with a persistent JSON-RPC connection to Codex**. We updated the spawn arguments to use `app-server`, introduced a single long-lived child process with new logic to send/receive JSON-RPC messages, adjusted how we parse output (methods/results instead of type/msg), and kept the outward API consistent. We also plan for config changes like mounting `CODEX_HOME` for auth and possibly using environment-provided API keys. After these changes, the proxy's external behavior (/v1/chat/completions streaming and non-streaming responses) should remain the same, but internally it will be driving the Codex app-server in place of the old proto-based workflow.

---

[1] [15] [26] [27] shared.js
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/src/handlers/chat/shared.js

[2] [3] [4] [5] [6] [13] [14] [18] [19] [20] [28] [29] stream.js
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/src/handlers/chat/stream.js

[7] server.js
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/server.js

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

9 utils.js
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/src/utils.js

10 11 25 32 33 fake-codex-proto.js
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/scripts/fake-codex-proto.js

12 23 24 Codex changelog
https://developers.openai.com/codex/changelog/

16 index.js
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/src/config/index.js

17 README.md
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/.codev/README.md

21 .env.dev.example
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/.env.dev.example

22 Dockerfile
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/Dockerfile

30 31 34 nonstream.js
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/src/handlers/chat/nonstream.js

35 package.json
https://github.com/DrJLabs/codex-completions-api/blob/9923a64a7fe23719cc8df44f87f8f94efb2d0d75/package.json