Advanced Configuration

Use these options when you need more control over providers, policies, and integrations. For a quick start, see Basic Config.

Profiles

Profiles let you save named sets of configuration values and switch between them from the CLI.

Profiles are experimental and may change or be removed in future releases.

Profiles are not currently supported in the Codex IDE extension.

Define profiles under `[profiles.<name>]` in `config.toml`, then run `codex --profile <name>`:

```
model = "gpt-5-codex"
approval_policy = "on-request"

[profiles.deep-review]
model = "gpt-5-pro"
model_reasoning_effort = "high"
approval_policy = "never"

[profiles.lightweight]
model = "gpt-4.1"
approval_policy = "untrusted"
```

To make a profile the default, add `profile = "deep-review"` at the top level of `config.toml`. Codex loads that profile unless you override it on the command line.

One-off overrides from the CLI

In addition to editing `~/.codex/config.toml`, you can override configuration for a single run from the CLI:

- Prefer dedicated flags when they exist (for example, `--model`).
- Use `-c` / `--config` when you need to override an arbitrary key.

Examples:

```
# Dedicated flag
codex --model gpt-5.2

# Generic key/value override (value is TOML, not JSON)
codex --config model='"gpt-5.2"'
codex --config sandbox_workspace_write.network_access=true
codex --config 'shell_environment_policy.include_only=["PATH","HOME"]'
```

Notes:

- Keys can use dot notation to set nested values (for example, `mcp_servers.context7.enabled=false`).
- `--config` values are parsed as TOML. When in doubt, quote the value so your shell doesn't split it on spaces.
- If the value can't be parsed as TOML, Codex treats it as a string.

Config and state locations

Codex stores its local state under `CODEX_HOME` (defaults to `~/.codex`).

Common files you may see there:

- `config.toml` (your local configuration)
- `auth.json` (if you use file-based credential storage) or your OS keychain/keyring
- `history.jsonl` (if history persistence is enabled)
- Other per-user state such as logs and caches

For authentication details (including credential storage modes), see Authentication. For the full list of configuration keys, see Configuration Reference.

Project root detection

Codex discovers project configuration (for example, `.codex/` layers and `AGENTS.md`) by walking up from the working directory until it reaches a "project root".

By default, Codex treats a directory containing `.git` as the project root. To customize this behavior, set `project_root_markers` in `config.toml`:

```toml
# Treat a directory as the project root when it contains any of these
        markers.
project_root_markers = [".git", ".hg", ".sl"]
```

Set `project_root_markers = []` to skip searching parent directories and treat the current working directory as the project root.

Custom model providers

A model provider defines how Codex connects to a model (base URL, wire API, and optional HTTP headers).

Define additional providers and point `model_provider` at them:

```toml
model = "gpt-5.1"
model_provider = "proxy"

[model_providers.proxy]
name = "OpenAI using LLM proxy"
base_url = "http://proxy.example.com"
env_key = "OPENAI_API_KEY"

[model_providers.ollama]
name = "Ollama"
base_url = "http://localhost:11434/v1"

[model_providers.mistral]
name = "Mistral"
base_url = "https://api.mistral.ai/v1"
env_key = "MISTRAL_API_KEY"
```

Add request headers when needed:

```toml
[model_providers.example]
http_headers = { "X-Example-Header" = "example-value" }
env_http_headers = { "X-Example-Features" = "EXAMPLE_FEATURES" }
```

OpenAI base URL override

If you just need to point the built-in OpenAI provider at an LLM proxy or router, set `OPENAI_BASE_URL` instead of defining a new provider. This overrides the default OpenAI endpoint without a `config.toml` change.

```
export OPENAI_BASE_URL="https://api.openai.com/v1"
```

`codex`

OSS mode (local providers)

Codex can run against a local "open source" provider (for example, Ollama or LM Studio) when you pass `--oss`. If you pass `--oss` without specifying a provider, Codex uses `oss_provider` as the default.

```toml
# Default local provider used with `--oss`
oss_provider = "ollama" # or "lmstudio"
```

Azure provider and per-provider tuning

```toml
[model_providers.azure]
name = "Azure"
base_url = "https://YOUR_PROJECT_NAME.openai.azure.com/openai"
env_key = "AZURE_OPENAI_API_KEY"
query_params = { api-version = "2025-04-01-preview" }
wire_api = "responses"

[model_providers.openai]
request_max_retries = 4
stream_max_retries = 10
stream_idle_timeout_ms = 300000
```

Model reasoning, verbosity, and limits

```toml
model_reasoning_summary = "none"          # Disable summaries
model_verbosity = "low"                   # Shorten responses
model_supports_reasoning_summaries = true # Force reasoning
model_context_window = 128000             # Context window size
```

`model_verbosity` applies only to providers using the Responses API. Chat Completions providers will ignore the setting.

Approval policies and sandbox modes

Pick approval strictness (affects when Codex pauses) and sandbox level (affects file/network access). See Sandbox & approvals for deeper examples.

```toml
approval_policy = "untrusted"   # Other options: on-request, on-
        failure, never
sandbox_mode = "workspace-write"

[sandbox_workspace_write]
exclude_tmpdir_env_var = false  # Allow $TMPDIR
exclude_slash_tmp = false       # Allow /tmp
writable_roots = ["/Users/YOU/.pyenv/shims"]
network_access = false          # Opt in to outbound network
```

In workspace-write mode, some environments keep `.git/` and `.codex/` read-only even when the rest of the workspace is writable. This is why commands like `git commit` may still require approval to run outside the sandbox. If you want Codex to skip specific commands (for example, block `git commit` outside the sandbox), use rules.

Disable sandboxing entirely (use only if your environment already isolates processes):

```toml
sandbox_mode = "danger-full-access"
```

Shell environment policy

`shell_environment_policy` controls which environment variables Codex passes to any subprocess it launches (for example, when running a tool-command the model proposes). Start from a clean slate (`inherit = "none"`) or a trimmed set (`inherit = "core"`), then layer on excludes, includes, and overrides to avoid leaking secrets while still providing the paths, keys, or flags your tasks need.

```toml
[shell_environment_policy]
inherit = "none"
set = { PATH = "/usr/bin", MY_FLAG = "1" }
ignore_default_excludes = false
exclude = ["AWS_*", "AZURE_*"]
include_only = ["PATH", "HOME"]
```

Patterns are case-insensitive globs (`*`, `?`, `[A-Z]`); `ignore_default_excludes = false` keeps the automatic KEY/SECRET/TOKEN filter before your includes/excludes run.

MCP servers

See the dedicated MCP documentation for configuration details.

Observability and telemetry

Enable OpenTelemetry (OTel) log export to track Codex runs (API requests, SSE/events, prompts, tool approvals/results). Disabled by default; opt in via `[otel]`:

```
[otel]
environment = "staging"   # defaults to "dev"
exporter = "none"         # set to otlp-http or otlp-grpc to send
        events
log_user_prompt = false   # redact user prompts unless explicitly
        enabled
```

Choose an exporter:

```
[otel]
exporter = { otlp-http = {
  endpoint = "https://otel.example.com/v1/logs",
  protocol = "binary",
  headers = { "x-otlp-api-key" = "${OTLP_TOKEN}" }
}}
```

```
[otel]
exporter = { otlp-grpc = {
  endpoint = "https://otel.example.com:4317",
  headers = { "x-otlp-meta" = "abc123" }
}}
```

If `exporter = "none"` Codex records events but sends nothing. Exporters batch asynchronously and flush on shutdown. Event metadata includes service name, CLI version, env tag, conversation id, model, sandbox/approval settings, and per-event fields (see Config Reference).

What gets emitted

Codex emits structured log events for runs and tool usage. Representative event types include:

- `codex.conversation_starts` (model, reasoning settings, sandbox/approval policy)
- `codex.api_request` and `codex.sse_event` (durations, status, token counts)
- `codex.user_prompt` (length; content redacted unless explicitly enabled)
- `codex.tool_decision` (approved/denied and whether the decision came from config vs user)
- `codex.tool_result` (duration, success, output snippet)

For more security and privacy guidance around telemetry, see Security.

Metrics

By default, Codex periodically sends a small amount of anonymous usage and health data back to OpenAI. This helps detect when Codex isn't working correctly and shows what features and configuration options are being used, so the Codex team can focus on what matters most. These metrics do not contain any personally identifiable information (PII).

If you want to disable metrics collection entirely across Codex surfaces on a machine, set the analytics flag in your config:

```
[analytics]
enabled = false
```

Each metric includes its own fields plus the default context fields below. You can use these fields to filter, group, or alert on metrics in your observability backend.

Default context fields (applies to every event/metric)

- `surface`: `cli` | `vscode` | `exec` | `mcp` | `subagent_*`.
- `version`: version number.
- `auth_mode`: `swic` | `api` | `unknown`.
- `model`: name of the model used.

Metrics catalog

Each metric includes the required fields plus the default context fields above. Every metric is prefixed by `codex.`. If a metric include the `tool` fields, this get populated by the internal tool used (`apply_patch`, `shell`, …) and does not contain the actual shell command or patch `codex` is trying to apply.

Metric	Type	Fields	Description
`features.state`	counter	`key`, `value`	Feature values that differ from defaults (emit one row per non-default).
`thread.started`	counter	`is_git`	New thread created.
`task.compact`	counter	`type`	Number of compactions per type (`remote` or `local`), including manual and au-

Metric	Type	Fields	Description
			to.
`task.user_shell`	counter		Number of user shell actions (`!` in the TUI for example).
`task.review`	counter		Number of reviews triggered.
`approval.requested`	counter	`tool`, `approved`	Tool approval request result (`approved`: `yes` or `no`).
`conversation.turn.count`	counter		User/assistant turns per thread, recorded at the end of the thread.
`mcp.call`	counter	`status`	MCP tool invocation result (`ok` or error string).
`model.call.duration_ms`	histogram	`status`, `attempt`	Model API request duration.
`tool.call`	counter	`tool`, `status`	Tool invocation result (`ok` or error string).
`tool.call.duration_ms`	histogram	`tool`, `success`	Tool execution time.
`user.feedback.submitted`	counter	`category`, `include_logs`, `success`	Feedback submission via `/feedback`.

Feedback controls

By default, Codex lets users send feedback from `/feedback`. To disable feedback collection across Codex surfaces on a machine, update your config:

```toml
[feedback]
enabled = false
```

When disabled, `/feedback` shows a disabled message and Codex rejects feedback submissions.

Hide or surface reasoning events

If you want to reduce noisy “reasoning” output (for example in CI logs), you can suppress it:

```
hide_agent_reasoning = true
```

If you want to surface raw reasoning content when a model emits it:

```
show_raw_agent_reasoning = true
```

Enable raw reasoning only if it's acceptable for your workflow. Some models/providers (like `gpt-oss`) do not emit raw reasoning; in that case, this setting has no visible effect.

Notifications

Use `notify` to trigger an external program whenever Codex emits supported events (currently only `agent-turn-complete`). This is handy for desktop toasts, chat webhooks, CI updates, or any side-channel alerting that the built-in TUI notifications don't cover.

```
notify = ["python3", "/path/to/notify.py"]
```

Example `notify.py` (truncated) that reacts to `agent-turn-complete`:

```
#!/usr/bin/env python3
import json, subprocess, sys

def main() -> int:
    notification = json.loads(sys.argv[1])
    if notification.get("type") != "agent-turn-complete":
        return 0
    title = f"Codex: {notification.get('last-assistant-message', 'Turn
        Complete!')}"
    message = " ".join(notification.get("input-messages", []))
    subprocess.check_output([
        "terminal-notifier",
        "-title", title,
        "-message", message,
        "-group", "codex-" + notification.get("thread-id", ""),
        "-activate", "com.googlecode.iterm2",
    ])
    return 0

if __name__ == "__main__":
    sys.exit(main())
```

The script receives a single JSON argument. Common fields include:

- `type` (currently `agent-turn-complete`)
- `thread-id` (session identifier)

- `turn-id` (turn identifier)
- `cwd` (working directory)
- `input-messages` (user messages that led to the turn)
- `last-assistant-message` (last assistant message text)

Place the script somewhere on disk and point `notify` to it.

`notify` vs `tui.notifications`

- `notify` runs an external program (good for webhooks, desktop notifiers, CI hooks).
- `tui.notifications` is built in to the TUI and can optionally filter by event type (for example, `agent-turn-complete` and `approval-requested`).

See Configuration Reference for the exact keys.

History persistence

By default, Codex saves local session transcripts under `CODEX_HOME` (for example, `~/.codex/history.jsonl`). To disable local history persistence:

```toml
[history]
persistence = "none"
```

To cap the history file size, set `history.max_bytes`. When the file exceeds the cap, Codex drops the oldest entries and compacts the file while keeping the newest records.

```toml
[history]
max_bytes = 104857600 # 100 MiB
```

Clickable citations

If you use a terminal/editor integration that supports it, Codex can render file citations as clickable links. Configure `file_opener` to pick the URI scheme Codex uses:

```toml
file_opener = "vscode" # or cursor, windsurf, vscode-insiders, none
```

Example: a citation like `/home/user/project/main.py:42` can be rewritten into a clickable `vscode://file/...:42` link.

Project instructions discovery

Codex reads `AGENTS.md` (and related files) and includes a limited amount of project guidance in the first turn of a session. Two knobs control how this works:

- `project_doc_max_bytes`: how much to read from each `AGENTS.md` file
- `project_doc_fallback_filenames`: additional filenames to try when `AGENTS.md` is missing at a directory level

For a detailed walkthrough, see Custom instructions with AGENTS.md.

TUI options

Running `codex` with no subcommand launches the interactive terminal UI (TUI). Codex exposes some TUI-specific configuration under `[tui]`, including:

- `tui.notifications`: enable/disable notifications (or restrict to specific types)
- `tui.animations`: enable/disable ASCII animations and shimmer effects
- `tui.scroll_*`: tune wheel/trackpad scroll behavior in the TUI2 viewport

See Configuration Reference for the full key list.