ChatGPT Projects CLI Documentation (v5)

Introduction

ChatGPT Projects CLI is a command-line interface (CLI) tool designed to facilitate isolated project management and AI-assisted coding workflows using OpenAI's GPT models. It emulates a "website-parity" experience similar to ChatGPT's web interface but with enhanced project isolation, artifact extraction, cost/token awareness, and **batch processing capabilities**. The tool ensures strict scoping: GPT interactions are limited to the current project's files, preventing accidental leakage of unrelated data.

Key principles:

- **Isolation**: Each project is a self-contained sandbox with files, artifacts, conversations, and metadata.
- **Transparency**: Token estimation, cost projection, and automatic artifact saving promote efficient usage.
- **Robustness**: Handles API variations (Responses vs. Chat Completions), streaming fallbacks, and error retries.
- **Extensibility**: Supports model switching, with aliases for common GPT variants and live API model listing.
- **Automation**: Batch mode enables scripted workflows for repetitive tasks and CI/CD integration.

This documentation is technical, focusing on engineering aspects, including token management, pricing calculations, model integration, and batch processing. It includes diagrams for architectural visualization.

Dependencies:

- openai (core API client)
- python-dotenv (for API key loading)
- tiktoken (optional, for precise token estimation; falls back to heuristic if unavailable)
- Standard libraries: os, json, cmd, shutil, sys, etc.

Install via:

```text
pip install --upgrade openai python-dotenv tiktoken
```

Set OPENAI_API_KEY in .env or environment.

Run interactively: `python ChatGPTProjects_v5.py`

Run batch mode: `python ChatGPTProjects_v5.py instructions.txt`

Features

- **Project Management**: Create/open projects, add/remove/list files (scoped to /files subdir).
- **Chat Integration**: Send queries to GPT with optional file context inclusion (-n for none, -s for selective).
- **Artifact Extraction**: Automatically detects and saves code blocks from responses as versioned files in /artifacts.
- **Conversation Logging**: Per-chat JSON files + consolidated project history for auditability.
- **Model Flexibility**: Aliases (e.g., gpt5 → gpt-5) and direct API model IDs; temperature control (0.2 default, adjustable).
- **Token & Cost Awareness**: Estimates via tiktoken (or char/4 heuristic); pricing table for USD projections.
- **Streaming Support**: Prefers Responses API for real-time output; auto-falls back to non-streaming or Chat Completions if restricted.
- **Multi-Line Input**: -m flag for extended prompts.
- **Export & Sync**: Export files/artifacts; auto-sync metadata with filesystem.
- **Batch Processing**: Execute commands from text files for automation and scripting.

Architecture

The system is built around a modular CLI (cmd.Cmd subclass) interacting with core classes: Project, ArtifactManager, and GPTClient. It uses a REPL-style loop for commands, with batch mode support for automated execution.

High-Level Architecture Diagram

Below is an ASCII art representation of the system's layered architecture:

```text
```

```
+------------------------+    +------------------------+
|     CLI Layer          |    |    Data Storage        |
| (ChatGPTCLI: cmd.Cmd)  |    | (JSON/Metadata/Files)  |
| - Commands: create,chat|    | - Project dirs         |
| - Prompt handling      |<--->| - conversations/      |
| - Settings persistence |    | - files/               |
| - Batch mode execution |    | - artifacts/           |
+------------------------+    +------------------------+
          ^
          |
          v
+------------------------+    +------------------------+
|   Business Logic       |    |   API Integration      |
| - Project: file mgmt   |<--->| (GPTClient: OpenAI)   |
| - ArtifactManager: code|    | - Responses/Chat Comps |
| - run_instruction_file |    | - Streaming fallbacks  |
|   extraction/saving    |    | - Model listing        |
+------------------------+    +------------------------+
```

- **CLI Layer**: User-facing REPL with command parsing (e.g., do_chat handles flags like -s) and batch mode runner.
- **Business Logic**: Manages state (e.g., Project ensures isolation via files_path) and batch processing logic.
- **API Integration**: Abstracts OpenAI calls, with robust error handling (e.g., no temperature for Responses API).
- **Data Storage**: Filesystem-based persistence (e.g., metadata.json tracks file additions).

Chat Workflow Diagram

A sequence diagram (in Mermaid-like markdown) illustrating a typical chat flow:

```
User -> ChatGPTCLI: chat -s file.py -- "Analyze code"
ChatGPTCLI -> Project: get_project_context(selected_files=["file.py"])
Project -> ChatGPTCLI: Context string (file contents)
ChatGPTCLI -> GPTClient: stream(model="gpt-5", system=INSTRUCTIONS, user=Context + Message)
GPTClient -> ChatGPTCLI: Streamed deltas (response text)
ChatGPTCLI -> User: Print deltas in real-time
ChatGPTCLI -> ArtifactManager: extract_code_blocks(response)
ArtifactManager -> ChatGPTCLI: Save artifacts to /artifacts
ChatGPTCLI -> Project: save_conversation(messages, response, artifacts)
Project -> ChatGPTCLI: Conversation saved
```

- **Flow Notes**: Context building checks token limits; artifacts are extracted via regex (triple-backticks with language).
- **Error Paths**: If streaming fails (e.g., 400 error), retry non-streaming; if Responses API fails, fallback to Chat Completions.

Project Directory Structure

An example tree diagram (ASCII art):

```text
ChatGPT_Projects/
├── my_project/                 # Project root
│   ├── files/                  # User-added files (isolated scope)
│   │   ├── script.py           # Added via 'add path/to/script.py'
│   │   └── data.json
│   ├── artifacts/              # Auto-extracted code from responses
│   │   ├── MyClass_20250808.py    # Versioned artifact
│   │   └── utils_20250808.js
│   ├── conversations/          # Per-chat logs
│   │   ├── conversation_20250808.json
│   │   └── project_history.json   # Consolidated history
│   └── metadata.json           # File metadata (added timestamps, sizes)
└── .settings.json              # CLI settings (last project, model)
```

- **Isolation Engineering**: All paths are relative to project_path; no global filesystem access.
- **Sync Mechanism**: sync_files() reconciles metadata with actual files (adds missing, removes deleted).

Usage Guide

Basic Workflow

1. `create my_project` → Creates isolated dir.
2. `add /path/to/file.py` → Copies file to /files.
3. `files` → Lists scoped files.
4. `chat "Improve this code"` → Sends with all files as context.
5. Response streams; code blocks saved as artifacts.
6. `artifacts` → View extracted files.
7. `history 5` → View last 5 chats in consolidated history.

Advanced Options

- Selective Context: `chat -s file1.py,file2.js -- "Refactor these"`
- No Context: `chat -n "General question"`
- Multi-Line: `chat -m` (type lines, end with EOF).
- Model Switch: `model 4o` (aliases: gpt5, 4.1, 4o, mini).
- Export: `export artifacts ~/backup` (copies to path).

Full command help: `help` in CLI.

Batch Mode Guide

Batch mode allows executing a sequence of CLI commands from a text file, enabling automation, scripting, and reproducible workflows.

Running Batch Mode

```bash
python ChatGPTProjects_v5.py instructions.txt
```

Instruction File Format

- **Comments**: Lines starting with `#` are ignored
- **Empty Lines**: Blank lines are skipped
- **Commands**: Each non-comment line is executed as if typed in the CLI
- **Multi-line Commands**: Use trailing backslash `\` for line continuation

Example Batch File

```bash
# Setup project and add files
create code_review_project
open code_review_project

# Add source files
add ./src/main.py
add ./src/utils.py
add ./tests/test_main.py

# Analyze the codebase
chat Please analyze this codebase and provide:\
   1. Architecture overview\
   2. Code quality assessment\
   3. Security vulnerabilities\
   4. Performance bottlenecks\
   5. Suggested improvements

# Generate documentation
chat -n Create comprehensive documentation for this project

# Check token usage
tokens

# Export results
export artifacts ./review_results

# Show history
history 5
```

Batch Mode Technical Details

Processing Mechanism

The batch processor uses a streaming approach rather than a queue:

1. **File Reading**: All lines loaded into memory via `splitlines()`
2. **Line Processing**: Sequential iteration with buffer for multi-line commands
3. **Command Execution**: Immediate execution via `cli.onecmd()`
4. **Error Handling**: Configurable stop-on-error behavior (default: true)

Implementation Flow

```python
buffer = []  # Temporary storage for multi-line commands

for idx, line in enumerate(lines):
    if line.endswith("\\"):
        buffer.append(line[:-1] + " ")  # Continue accumulating
    else:
        buffer.append(line)
        command = "".join(buffer).strip()
        cli.onecmd(command)  # Execute immediately
        buffer.clear()
```

Error Handling

- **Default**: Stops on first error (`stop_on_error=True`)
- **Continue Mode**: Set `stop_on_error=False` to log errors but continue
- **Exit Commands**: `exit` or `quit` in batch file stops execution
- **Echo Mode**: Commands are printed before execution for visibility

Use Cases

1. **Automated Code Reviews**: Batch analyze multiple files with consistent prompts
2. **Project Setup**: Standardize project initialization across teams
3. **CI/CD Integration**: Integrate AI analysis into build pipelines
4. **Reproducible Workflows**: Share instruction files for consistent results
5. **Bulk Operations**: Process multiple projects with similar operations

Best Practices

- Start batch files with project creation/opening
- Use comments liberally for documentation
- Test commands interactively before batch execution
- Include error recovery commands where appropriate
- Save batch files with descriptive names (e.g., `review_python_project.txt`)

Technical Details

Token Limits and Estimation

- **Global Limit**: Set to MAX_INPUT_TOKENS = 272,000 and MAX_TOTAL_TOKENS = 400,000 (based on GPT-5 context window).
- **Estimation Engine**: Prefers tiktoken for precise encoding (o200k_base for modern models). Fallback: len(text) / 4 heuristic (conservative; assumes ~4 chars/token).
- **Budget Checks**: Before chat, sums context + message + overhead (2000 tokens). If > limit - 5000, aborts. Selective inclusion (-s) allows manual trimming.
- **Overhead Breakdown**:
  - System prompt: ~200 tokens.
  - File wrappers: ~50 tokens per file (e.g., "--- File: name ---").
  - Response buffer: Reserved for output.
- **Visualization**: `tokens` command outputs a table with per-file tokens, colored icons (🟢 <20k, 🟡 <50k, 🔴 >50k), and total projection.

Engineering Note: Token checks prevent API errors; future enhancements could include auto-truncation.

Pricing Calculations

- **Table**: PRICING_USD_PER_1K dict (input/output rates per model). Example:
  - GPT-5: $0.00 input / $0.00 output (placeholder; update post-release).
  - GPT-4o: $5.00 input / $15.00 output.
- **One-Shot Estimation**: (in_tokens / 1000) * input_rate + (out_tokens / 1000) * output_rate.
  - Defaults: 50% input ratio, 200k total tokens.
- **Hourly Projection**: Assumes throughput (e.g., 30 tokens/sec), balanced input/output: tps * 3600 * (avg_rate / 1000).
- **Command**: `pricing --model 4o --estimate-tokens 100000 --in-ratio 0.6 --throughput-tps 50`.
- **Accuracy**: Based on estimates; actual costs vary by API. No real-time billing query (API limitation).

Engineering Note: Pricing is static; integrate OpenAI's usage API for post-chat actuals in future versions.

Model Choice and Integration

- **Aliases**: Mapped in MODELS dict (e.g., mini → gpt-4o-mini). Custom IDs allowed via `model exact-id`.
- **Listing**: `models` pulls live from client.models.list() (abbreviated to 30 for brevity).

- **API Preference**: Responses API first (for website-like behavior; no temperature sent to avoid 400s). Fallback to Chat Completions if unavailable or restricted.
- **Temperature**: 0.25 default (creativity balance); sent only to Chat Completions.
- **Streaming Logic**:
  - Try Responses streaming.
  - On failure (e.g., "stream not allowed"), retry non-stream.
  - On content-type errors, switch to instructions param.
  - Ultimate fallback: Chat Completions (streaming or non).
- **System Prompt**: Fixed (SYSTEM_INSTRUCTIONS + website-style guidelines) for consistent engineering-focused responses.

Engineering Note: Robustness via try-except chains; handles org-level restrictions (e.g., no streaming for some plans).

Limitations & Tips

- **Limits**: No image/multimodal support (text-only). Token overflows require manual intervention.
- **Performance**: Streaming may lag on slow networks; non-stream fallback ensures completion.
- **Security**: Files are copied (not linked); avoid sensitive data.
- **Batch Mode**: Commands execute sequentially; no parallel processing. Syntax errors in batch files may cause unexpected behavior.
- **Tips**:
  - Monitor tokens before large chats.
  - Use -s for efficiency in big projects.
  - Update pricing table for new models.
  - For artifacts: Regex-based extraction skips small snippets (<100 chars).
  - Test batch files with small projects first.
  - Use version control for instruction files.
- **Extending**: Add custom commands by subclassing ChatGPTCLI; integrate other LLMs via adapter pattern.

This tool bridges CLI simplicity with AI power, engineered for reliability in coding workflows. Batch mode extends this with automation capabilities for scalable AI-assisted development. Contributions welcome!