Claude Desktop Client

Product Requirements Document (PRD)

High-Level & Low-Level Design Specification

Version: 1.0

Date: 2026-02-15

Target Platform: Windows 10/11 Desktop

Tech Stack: Python (PyQt5/PySide6)

API: Anthropic Claude Messages API

PART ONE: HIGH-LEVEL PRD

1. Product Overview

1.1 Product Name

Claude Desktop Client (暂定名: ClaudeStation)

1.2 Product Vision

Build a Windows desktop application that connects to the Anthropic Claude API, providing a feature set equivalent to Claude.ai's Projects functionality. The application enables users to manage multiple projects with shared document context, conduct multiple conversations within each project, select from all Claude 4.5+ models, and maximize token savings through prompt caching and intelligent context management.

1.3 Target Users

- Developers and researchers who need to interact with Claude via API while managing complex, multi-document projects
- Power users who want full control over model selection, token budget, and conversation management
- Users who need offline-capable document management with cloud-based AI inference

1.4 Core Value Propositions

1. Project-based organization: Replicate Claude.ai Projects with local document uploads and multiple conversations per project
2. Model flexibility: Support all Claude 4.5+ models (Opus 4.5, Sonnet 4.5, Haiku 4.5, and Opus 4.6) with real-time switching
3. Token economy: Aggressive token savings via prompt caching, context compression, and smart truncation
4. Full data sovereignty: All project data stored locally; only API calls leave the machine

2. Feature Matrix

The following table maps Claude.ai Projects features to the desktop client's implementation plan:

Claude.ai Feature	Desktop Implementation	Priority
Projects	Local project folders with metadata JSON	P0 - Must Have

Project Knowledge (file uploads)	Local file ingestion (PDF, DOCX, TXT, MD, CSV, code files) with text extraction	P0 - Must Have
Project Instructions (system prompt)	Custom system prompt editor per project with template support	P0 - Must Have
Multiple conversations per project	Conversation list with rename, archive, delete; each conversation is independent	P0 - Must Have
Model selection	Dropdown: Opus 4.6, Opus 4.5, Sonnet 4.5, Haiku 4.5; per-conversation override	P0 - Must Have
Streaming responses	SSE streaming with real-time markdown rendering	P0 - Must Have
Prompt Caching	Automatic cache_control on system prompt + project docs; cache hit/miss monitoring	P0 - Must Have
Token usage tracking	Per-conversation and per-project token/cost dashboard	P0 - Must Have
Extended Thinking	Toggle + budget slider for thinking tokens	P1 - Should Have
Conversation search	Full-text search across all conversations in a project	P1 - Should Have
Export/Import	Export conversations as Markdown/JSON; import project bundles	P1 - Should Have
Image input	Paste/upload images as message attachments (base64 encoding)	P1 - Should Have
Tool Use	Custom tool definition editor with JSON schema support	P2 - Nice to Have

3. Token Savings Strategy

Token cost optimization is a first-class requirement. The following strategies work together to minimize API spend:

3.1 Prompt Caching (Primary Savings: up to 90%)

Prompt caching is the single most impactful optimization. The system prompt and project documents are placed at the beginning of every API request as cached content. On subsequent calls within the same session, these tokens are read from cache at only 10% of the base input price.

- Cache write cost: 1.25x base input price (one-time per cache creation)
- Cache read cost: 0.1x base input price (every subsequent call)
- Cache TTL: 5 minutes by default, 1 hour with extended TTL (supported on 4.5 models)
- Minimum cacheable prefix: 1,024 tokens
- Strategy: Place system prompt + all project documents as the first content blocks, mark with cache_control, then append conversation history and user message after the cache breakpoint

3.2 Intelligent Context Window Management

- Conversation truncation: When conversation history approaches the context window limit, automatically summarize older turns and keep only the most recent N turns in full
- Document relevance filtering: Use keyword matching or embedding similarity to include only relevant documents from the project knowledge base, rather than sending all documents every time
- Lazy document loading: Only include documents that are explicitly referenced or semantically relevant to the current conversation turn

3.3 Model Routing

Allow users to set a "default model" per project and override per-conversation. For simple Q&A, route to Haiku 4.5 ($1/$5 per MTok); for complex reasoning, use Sonnet 4.5 ($3/$15) or Opus 4.5 ($5/$25). Provide a cost estimator before sending.

3.4 Effort Control

Claude Opus 4.5 supports an effort parameter that can reduce output token usage by up to 76% while maintaining quality. Expose this as a slider: Low (fast/cheap), Medium (balanced), High (max quality).

4. System Architecture Overview

The application follows a layered architecture with clear separation of concerns:

Layer	Components	Responsibility
Presentation (UI)	PyQt5/PySide6 widgets, QWebEngineView for markdown	User interaction, rendering, input handling
Application Logic	ProjectManager,	Business logic, state

	ConversationManager, CacheManager	management, token optimization
API Layer	ClaudeAPIClient (anthropic Python SDK)	API calls, streaming, error handling, retry logic
Data Layer	SQLite + file system	Project/conversation persistence, document storage, settings

5. Non-Functional Requirements

5.1 Performance

- First token latency: < 2 seconds with cached prompts
- UI responsiveness: All UI operations complete in < 100ms; streaming updates at 60fps
- Document indexing: Process and extract text from a 100-page PDF in < 10 seconds

5.2 Security

- API key stored encrypted using Windows DPAPI (via keyring library)
- All project data stored locally; no telemetry or analytics sent externally
- HTTPS-only communication with api.anthropic.com

5.3 Compatibility

- Windows 10 (1809+) and Windows 11
- Python 3.10+ runtime
- Display scaling: 100%, 125%, 150%, 200% DPI

5.4 Data Storage

- Default data directory: ~/ClaudeStation/ (user configurable)
- SQLite database for metadata, conversation history, token usage tracking
- File system for uploaded documents (original + extracted text cache)

PART TWO: LOW-LEVEL DESIGN

6. Project Directory Structure

The Python project should follow this layout:

```
claude_station/
  main.py                      # Application entry point
  config.py                    # Global configuration & constants
  requirements.txt               # Dependencies

  ui/                        # Presentation layer
    main_window.py              # Main window with sidebar + content area
    project_panel.py            # Project list and creation dialog
    conversation_panel.py       # Conversation list within a project
    chat_widget.py              # Chat messages display (Markdown rendered)
    input_widget.py             # Message input with file attach button
    settings_dialog.py          # API key, model defaults, cache settings
    token_dashboard.py          # Token usage visualization
    document_manager.py         # Upload, preview, remove project docs
    model_selector.py           # Model dropdown with cost info
    system_prompt_editor.py     # Project-level system prompt editor

  core/                      # Application logic layer
    project_manager.py          # CRUD for projects
    conversation_manager.py     # CRUD for conversations, history management
    cache_manager.py            # Prompt caching logic & cache_control injection
    context_builder.py          # Build API request: system + docs + history + user msg
    document_processor.py       # Extract text from PDF, DOCX, code, etc.
    token_counter.py            # Token counting using tiktoken/anthropic tokenizer
    token_tracker.py            # Usage tracking and cost calculation
    context_compressor.py       # Summarize old turns, truncate intelligently
  api/                      # API layer
    claude_client.py          # Anthropic SDK wrapper with streaming
    models.py                 # Model definitions, pricing, context windows
    error_handler.py          # Retry logic, rate limiting, error UI feedback

  data/                     # Data layer
    database.py               # SQLite connection, migrations
    schemas.py                # Table definitions
    file_storage.py           # Document file management

  utils/                    # Utilities
    crypto.py                 # API key encryption/decryption
    markdown_renderer.py      # MD to HTML conversion
    export.py                 # Export conversations
```

7. Database Schema (SQLite)

All metadata is persisted in a single SQLite database file. The schema consists of four main tables:

7.1 projects table

Column	Type	Nullable	Description
id	TEXT PK	NO	UUID v4
name	TEXT	NO	Project display name
system_prompt	TEXT	YES	Custom system prompt (Project Instructions)
default_model	TEXT	NO	Default model ID (e.g. claude-sonnet-4-5-20250929)
created_at	DATETIME	NO	Creation timestamp
updated_at	DATETIME	NO	Last modified timestamp
settings_json	TEXT	YES	JSON blob: effort level, thinking budget, max context tokens, etc.

7.2 documents table

Column	Type	Nullable	Description
id	TEXT PK	NO	UUID v4
project_id	TEXT FK	NO	References projects.id
filename	TEXT	NO	Original filename
file_path	TEXT	NO	Relative path in storage
extracted_text	TEXT	YES	Extracted plain text content
token_count	INTEGER	YES	Token count of extracted text
file_type	TEXT	NO	MIME type or extension
created_at	DATETIME	NO	Upload timestamp

7.3 conversations table

Column	Type	Nullable	Description
id	TEXT PK	NO	UUID v4
project_id	TEXT FK	NO	References projects.id
title	TEXT	NO	Auto-generated or user-set title
model_override	TEXT	YES	Override project default model
created_at	DATETIME	NO	Creation timestamp
updated_at	DATETIME	NO	Last message timestamp
is_archived	BOOLEAN	NO	Soft delete / archive flag

7.4 messages table

Column	Type	Nullable	Description
id	TEXT PK	NO	UUID v4
conversation_id	TEXT FK	NO	References conversations.id
role	TEXT	NO	"user" or "assistant"
content	TEXT	NO	Message text content

thinking_content	TEXT	YES	Extended thinking output (if enabled)
attachments_json	TEXT	YES	JSON: attached images (base64 refs)
model_used	TEXT	YES	Actual model used for this response
input_tokens	INTEGER	YES	Input tokens consumed
output_tokens	INTEGER	YES	Output tokens generated
cache_read_tokens	INTEGER	YES	Tokens read from cache
cache_creation_tokens	INTEGER	YES	Tokens written to cache
cost_usd	REAL	YES	Calculated cost in USD
created_at	DATETIME	NO	Timestamp

8. Core Module Specifications

8.1 ClaudeAPIClient (api/claude_client.py)

This is the central API communication module. It wraps the official anthropic Python SDK.

8.1.1 Key Methods

send_message(messages, system, model, stream=True, cache_control=None, thinking=None)

- Sends a messages API request to Anthropic
- Parameters: messages (list of dicts), system (list of content blocks), model (string), stream (bool), cache_control (dict or None), thinking (dict with type and budget_tokens)
- Returns: Generator yielding streaming events if stream=True, or full response dict
- Implements exponential backoff retry (3 attempts, 1s/2s/4s delays) for 429 and 5xx errors

API Request Structure (Pseudocode):

```
client.messages.create(
    model = selected_model_id,
    max_tokens = 8192,  # configurable
    system = [
        {"type": "text", "text": system_prompt + "\n\n" + all_docs_text,
         "cache_control": {"type": "ephemeral"}}
    ],
    messages = conversation_history + [{"role": "user", "content": user_input}],
    stream = True
)
```

8.2 ContextBuilder (core/context_builder.py)

Responsible for assembling the full API request payload with optimal caching.

8.2.1 Build Logic

1. Assemble system prompt: Concatenate project's custom system prompt with all active project documents' extracted text. Each document is wrapped in XML tags: <document name="filename.pdf">extracted text</document>
2. Apply cache_control: Set {"type": "ephemeral"} on the system content block to enable prompt caching. For sessions longer than 5 minutes, use {"type": "ephemeral", "ttl": "1h"} on 4.5 models.
3. Build message history: Include the last N turns from the conversation. If total tokens exceed 80% of the model's context window, trigger the ContextCompressor.
4. Append user message: Add the current user input as the final message. If the user attached an image, include it as a base64 content block.
5. Token budget check: Use token_counter to estimate total tokens. If over limit, compress further or warn user.

8.3 CacheManager (core/cache_manager.py)

Monitors and optimizes prompt caching performance.

- Tracks cache_creation_input_tokens and cache_read_input_tokens from each API response
- Calculates cache hit rate = cache_read_tokens / (cache_read_tokens + cache_creation_tokens)
- Alerts user if cache hit rate drops below 50% (indicates prompt changes are invalidating cache)
- Provides session-level cache savings report: (full_price - cached_price) / full_price

8.4 ContextCompressor (core/context_compressor.py)

Handles intelligent context window management to prevent exceeding model limits while preserving conversation quality.

8.4.1 Compression Strategies (in order of application)

1. Sliding window: Keep only the most recent K message pairs (default K=20). Archive older messages to local database.
2. Summary injection: When truncating, generate a summary of the truncated portion using a cheap model (Haiku 4.5) and prepend it as a system message: "[Summary of previous conversation: ...]"
3. Document pruning: If project has many documents but the current conversation only references a subset, only include referenced documents in the system prompt. Use keyword matching against the last 3 user messages.
4. Max context allocation: Reserve at least 4096 tokens for the model's response. The remaining budget is split: 60% for system+docs (cached), 40% for conversation history.

8.5 DocumentProcessor (core/document_processor.py)

Extracts plain text from various document formats for inclusion in the API context.

8.5.1 Supported Formats and Libraries

Format	Python Library	Notes
PDF	PyMuPDF (fitz) or pdfplumber	Handles text extraction, tables, OCR fallback
DOCX	python-docx	Extracts paragraphs, tables, headers/footers
TXT/MD/CSV	Built-in open()	Direct read with encoding detection (chardet)
Code files (.py, .js, etc.)	Built-in open()	Read with syntax-aware truncation
XLSX	openpyxl	Convert sheets to CSV-like text
Images (PNG, JPG)	base64 + PIL	Sent as vision content blocks, not extracted text

8.6 TokenTracker (core/token_tracker.py)

Records all token usage from API responses and calculates costs.

8.6.1 Cost Calculation Formula

For each API response, compute:

```
cost = (input_tokens * input_price / 1_000_000)
```

```
+ (output_tokens * output_price / 1_000_000)
+ (cache_creation_tokens * input_price * 1.25 / 1_000_000)
+ (cache_read_tokens * input_price * 0.1 / 1_000_000)
```

8.6.2 Model Pricing Table (to be maintained in models.py)

Model	Input $/MTok	Output $/MTok	Context Window	Model ID
Claude Opus 4.6	$5.00	$25.00	200K	claude-opus-4-6
Claude Opus 4.5	$5.00	$25.00	200K	claude-opus-4-5-20251101
Claude Sonnet 4.5	$3.00	$15.00	200K (1M beta)	claude-sonnet-4-5-20250929
Claude Haiku 4.5	$1.00	$5.00	200K	claude-haiku-4-5-20251001

9. UI Layout Specification

9.1 Main Window Layout

The main window uses a three-panel layout:

Left Sidebar (240px, collapsible):

- Project list with + button for new project
- Below project list: conversation list for selected project
- Each conversation shows title, last message preview, timestamp
- Right-click context menu: Rename, Archive, Delete, Export

Center Panel (flexible width):

- Chat message area: scrollable, with user messages right-aligned, assistant messages left-aligned
- Each message shows: avatar, content (rendered Markdown with code highlighting), timestamp, token count badge
- Streaming indicator: animated dots during response generation

- Thinking content: collapsible section above the response, styled with a different background

Right Panel (280px, collapsible, toggle button):

- Project settings: system prompt editor (resizable text area)
- Documents list: uploaded files with token count, remove button, drag-and-drop upload area
- Token dashboard: session cost, total project cost, cache hit rate, pie chart of input/output/cache tokens

Top Bar:

- Model selector dropdown with cost indicator (e.g., "Sonnet 4.5 - $3/$15")
- Effort slider (Low / Medium / High) - only shown when Opus model selected
- Extended thinking toggle + budget input
- Settings gear icon

Bottom Bar:

- Message input: multi-line text area with Ctrl+Enter to send
- Attach file button (for in-message images)
- Send button
- Token estimate label: shows estimated input tokens before sending

10. API Integration Details

10.1 Authentication

The API key is entered once in Settings and stored encrypted via the keyring library (uses Windows Credential Manager under the hood). On each API call, the key is decrypted in memory. The anthropic Python SDK is initialized with: client = anthropic.Anthropic(api_key=decrypted_key).

10.2 Streaming Implementation

Use the SDK's streaming interface for real-time response display:

```
with client.messages.stream(...) as stream:
    for event in stream:
```

```
    if event.type == 'content_block_delta':
        yield event.delta.text      # Emit to UI thread via signal
    elif event.type == 'message_start':
        # Extract usage info
    elif event.type == 'message_stop':
        # Finalize, save to DB
```

The streaming must run in a QThread (or asyncio with qasync) to avoid blocking the UI. Use Qt signals to push text deltas to the chat widget.

10.3 Error Handling

Error Code	Cause	Handling
401	Invalid API key	Prompt user to re-enter key in Settings
429	Rate limited	Exponential backoff: 1s, 2s, 4s. Show countdown in UI
500/529	Server error / overloaded	Retry up to 3 times. If persistent, show error with "retry" button
Context too long	Tokens exceed model max	Auto-compress context and retry. If still too long, prompt user to remove documents or start new conversation
Network error	No internet / DNS failure	Show offline indicator. Queue message for retry when online

11. Prompt Caching Implementation Details

This section provides the exact caching strategy for the developer implementing the application.

11.1 Cache Structure

Every API call must structure the request as follows to maximize cache hits:

Layer 1 (System Prompt - cached):

The system prompt is the outermost cached layer. It includes the project's custom instructions and all project documents. This content changes infrequently (only when user edits system prompt or uploads/removes documents). Mark the entire system block with cache_control.

Layer 2 (Conversation History - not cached):

The messages array contains the conversation history. This changes with every turn, so it is NOT cached. However, the system prompt cache from Layer 1 is still valid and will produce cache hits.

Layer 3 (Current User Message - not cached):

The new user message is appended at the end.

11.2 Cache Optimization Rules

1. Never modify the system prompt content between turns in the same session. Even whitespace changes invalidate the cache.
2. When the user uploads a new document mid-session, regenerate the system prompt (cache miss expected) and continue. The new system prompt will be cached for subsequent turns.
3. Use 1-hour TTL (ttl: "1h") for long work sessions. This is supported on Claude Opus 4.5, Sonnet 4.5, and Haiku 4.5.
4. Monitor cache_read_input_tokens in every response. If it drops to 0 unexpectedly, log a warning and investigate what changed.
5. Pre-warm the cache: When user opens a project, send a lightweight "hello" message to create the cache entry before the user starts typing.

12. Python Dependencies (requirements.txt)

```
anthropic>=0.40.0          # Official Anthropic Python SDK
PyQt5>=5.15 or PySide6     # GUI framework
tiktoken                   # Token counting (fallback)
PyMuPDF>=1.24.0            # PDF text extraction
python-docx>=1.1.0         # DOCX text extraction
openpyxl>=3.1.0            # XLSX text extraction
chardet>=5.0               # Encoding detection
keyring>=25.0              # Secure API key storage
Pillow>=10.0               # Image handling
markdown>=3.5              # Markdown to HTML
```

```
pygments>=2.17          # Code syntax highlighting
qasync>=0.27          # Qt + asyncio integration
```

13. Development Phases

Phase 1: Core MVP (Week 1-2)

1. Project CRUD + SQLite persistence
2. Single conversation with streaming API calls
3. Model selection dropdown
4. Basic chat UI with Markdown rendering
5. API key management with keyring

Phase 2: Projects Feature (Week 3-4)

6. Document upload and text extraction (PDF, DOCX, TXT)
7. System prompt editor per project
8. Multiple conversations per project
9. Prompt caching with cache_control injection
10. Token usage tracking and cost display

Phase 3: Optimization (Week 5-6)

11. Context compression (sliding window + summarization)
12. Extended thinking toggle
13. Cache hit rate monitoring dashboard
14. Image attachment support
15. Conversation search and export

Phase 4: Polish (Week 7-8)

16. Effort control slider
17. Document relevance filtering
18. Dark/light theme
19. Keyboard shortcuts
20. Error recovery and offline resilience

14. Key Implementation Notes for Developer

The following notes are critical for whoever implements this application:

1. Threading Model: All API calls MUST run in a background thread (QThread or asyncio). Never call the Anthropic SDK on the main Qt event loop thread. Use Qt signals/slots to push streaming text to the UI.

2. Token Counting: Use the anthropic SDK's built-in token counter (anthropic.count_tokens) if available, otherwise fall back to tiktoken with the cl100k_base encoding. Count tokens BEFORE sending to pre-validate context length.

3. Prompt Caching is Stateless: Prompt caching is NOT a session feature on Anthropic's servers. It works by hashing the prompt prefix. As long as you send the identical prefix (byte-for-byte), you get a cache hit. There is no session ID or cache key to manage. Just keep your system prompt + docs identical across calls.

4. Model ID Strings: Use exact model ID strings from Anthropic's documentation. Do not guess or abbreviate. The current valid IDs are: claude-opus-4-6, claude-opus-4-5-20251101, claude-sonnet-4-5-20250929, claude-haiku-4-5-20251001. These may change; consider fetching available models from the API at startup.

5. System Prompt Structure: The system parameter in the Messages API accepts a list of content blocks, not a plain string. To use caching, you must send: system=[{"type": "text", "text": "...", "cache_control": {"type": "ephemeral"}}]. A plain string system prompt cannot be cached.

6. Extended Thinking: When enabling extended thinking, set thinking={"type": "enabled", "budget_tokens": N} in the API call. The minimum budget is 1,024 tokens. Thinking tokens are billed as output tokens. The thinking content comes as separate content blocks in the response (type="thinking").

7. Encoding: All text must be UTF-8. When extracting text from documents, always normalize to UTF-8 and handle BOM characters. The Anthropic API expects UTF-8 encoded strings.

8. Cost Display: Always show cost in a non-alarming way. Display per-message cost and running session total. Use color coding: green (< $0.01), yellow ($0.01 - $0.10), orange (> $0.10) per message.