Claude Desktop Client PRD

Clarification Response Document

Addressing All Five Dimensions of Developer Questions

Date: 2026-02-15

Status: Final Decisions - Ready for Implementation

Dimension 1: Model & API Reality Confirmation

> CRITICAL CORRECTION: The developer's assumption that these are future models is WRONG. All models listed in the PRD are CURRENT and LIVE on the Anthropic API as of February 2026.

1.1 Model Version Clarification

The developer appears to have knowledge limited to the Claude 3 era (early-mid 2024). The Claude model family has evolved significantly since then. Here is the current model timeline:

Model	Release Date	API Model ID	Status
Claude 3 Opus	Mar 2024	claude-3-opus-20240229	Legacy (still available)
Claude 3.5 Sonnet	Jun 2024	claude-3-5-sonnet-20241022	Legacy
Claude 3.7 Sonnet	Feb 2025	claude-3-7-sonnet-20250219	Legacy
Claude Sonnet 4	May 2025	claude-sonnet-4-20250514	Active
Claude Opus 4	May 2025	claude-opus-4-20250514	Active
Claude Opus 4.1	Aug 2025	claude-opus-4-1-20250805	Active
Claude Sonnet 4.5	Sep 2025	claude-sonnet-4-5-20250929	Active (Recommended)
Claude Haiku 4.5	Oct 2025	claude-haiku-4-5-20251001	Active (Recommended)
Claude Opus 4.5	Nov 2025	claude-opus-4-5-20251101	Active (Recommended)
Claude Opus 4.6	Jan 2026	claude-opus-4-6	Active (Latest, Most Capable)

> DECISION: Use the 4.5 series (Opus 4.5/4.6, Sonnet 4.5, Haiku 4.5) as primary models. Optionally expose older models for cost-sensitive use cases.

1.2 Prompt Caching API Status

Prompt caching is NOT a beta feature. It is fully GA (Generally Available) on the Anthropic API for all current models. No special beta header is required.

Implementation: Simply include the cache_control field in the system content block. The API recognizes it natively:

```
system=[{"type": "text", "text": "...", "cache_control": {"type": "ephemeral"}}]
```

For 1-hour TTL (supported on 4.5 models), the anthropic Python SDK handles this transparently. No beta headers needed.

1.3 Pricing Confirmation

The pricing in the PRD is CORRECT for the 4.5 series. Here is the confirmed pricing as of February 2026:

Model	Input $/MTok	Output $/MTok	Cache Read $/MTok
Opus 4.5 / 4.6	$5.00	$25.00	$0.50 (10%)
Sonnet 4.5	$3.00	$15.00	$0.30 (10%)
Sonnet 4.5 (>200K input)	$6.00	$22.50	$0.60 (10%)
Haiku 4.5	$1.00	$5.00	$0.10 (10%)

Cache write cost is 1.25x the base input price. Cache read cost is 0.1x. The old Claude 3 Opus pricing ($15/$75) is irrelevant to this project.

> **Action for Developer:** Install anthropic>=0.40.0 via pip, create a test script calling client.messages.create() with any of the above model IDs. If the developer's API key doesn't have access to Claude 4.5 models, they need to upgrade their Anthropic account tier or contact Anthropic.

Dimension 2: Technical Stack Decisions

2.1 GUI Framework: PySide6

> DECISION: Use PySide6 (not PyQt5).

Reasons:

1. LGPL license avoids GPL contamination, allowing future commercial distribution if desired.
2. PySide6 is the official Qt for Python binding maintained by The Qt Company.
3. API is nearly identical to PyQt5 (95%+ compatible), so most online PyQt5 examples can be adapted with minimal changes (primarily import statements).
4. Better long-term support: PySide6 tracks Qt 6.x releases directly.

Migration note: Replace all 'from PyQt5.QtWidgets import ...' with 'from PySide6.QtWidgets import ...'. Signal syntax changes from pyqtSignal to Signal. That's essentially the only difference.

2.2 Python Version: 3.11+

> DECISION: Target Python 3.11 as minimum, develop on 3.12.

Python 3.11 provides significant performance improvements (10-60% faster than 3.10) and better error messages. Python 3.12 adds further asyncio improvements. The anthropic SDK and PySide6 both fully support 3.11 and 3.12.

2.3 Async Architecture: QThread + asyncio Hybrid

> DECISION: Use QThread as the primary threading model, with asyncio only inside the API client.

Rationale and implementation:

Main Thread (Qt Event Loop): All UI rendering, user input handling, and widget updates. Never perform I/O on this thread.

API Worker Thread (QThread): Runs a dedicated asyncio event loop for API calls. The

anthropic SDK's async client (anthropic.AsyncAnthropic) runs inside this loop. Communication with the main thread uses Qt signals:

```
class APIWorker(QThread):
    text_received = Signal(str)      # Streaming text delta
    response_complete = Signal(dict)  # Final response with usage
    error_occurred = Signal(str)      # Error messages

    def run(self):
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        loop.run_until_complete(self._stream_response())
```

Why NOT qasync: qasync integrates asyncio into Qt's event loop on the main thread, which can cause subtle UI freezes during DNS resolution or TLS handshake. A separate QThread with its own asyncio loop is more robust and easier to debug. Remove qasync from requirements.txt.

2.4 Code Standards

> DECISION: PEP 8 + mandatory type hints + docstrings.

1. PEP 8 compliance enforced via ruff or flake8.
2. Type hints required on all function signatures and return types. Use 'from __future__ import annotations' for modern syntax.
3. Google-style docstrings on all public methods.
4. No wildcard imports. Explicit imports only.
5. Line length: 100 characters max (not 79, for readability with type hints).

Dimension 3: Data Security & Deployment

3.1 Proxy Support: YES, Required

> DECISION: Build in HTTP/HTTPS proxy configuration from Day 1.

Implementation in Settings dialog:

- Proxy type dropdown: None / HTTP / HTTPS / SOCKS5
- Host and port fields
- Optional authentication (username/password)
- "Test Connection" button that pings api.anthropic.com through the proxy

In code, pass the proxy to the anthropic client:

```
import httpx
client = anthropic.Anthropic(
    api_key=key,
    http_client=httpx.Client(proxy="http://proxy:port")
)
```

The anthropic SDK uses httpx internally, which natively supports proxies. Also respect the system environment variables HTTP_PROXY / HTTPS_PROXY as fallback.

3.2 Multi-Account: YES, Support Multiple API Keys

> DECISION: Support multiple API key profiles.

Add an 'api_keys' table to the database:

Column	Type	Description
id	TEXT PK	UUID
label	TEXT	Display name (e.g., "Personal", "Work - Company X")
key_ref	TEXT	Reference ID for keyring storage (not the key itself)
is_default	BOOLEAN	Whether this is the default key for new projects

created_at	DATETIME	Creation timestamp

Each project can be associated with a specific API key profile. The actual API keys are stored in Windows Credential Manager via keyring, never in SQLite.

3.3 Data Backup

> DECISION: Manual export/import of entire data directory. No automatic backup.

- Provide a "Backup" button in Settings that creates a ZIP of the entire ~/ClaudeStation/ directory (excluding API keys which stay in Credential Manager).
- Provide an "Import Backup" button that unzips and merges/replaces the data directory.
- The SQLite database file can be copied directly as a portable backup.
- Individual project export/import via JSON bundles (includes conversations, documents, settings).

3.4 API Key Memory Safety

> DECISION: Standard keyring approach. No aggressive in-memory scrubbing.

The API key is decrypted from keyring into a Python string when the application starts (or when the user switches API key profiles). It remains in memory for the lifetime of the application. We do NOT implement lock-screen scrubbing because:

1. Python strings are immutable and cannot be reliably zeroed in memory.
2. The anthropic client object holds the key internally anyway.
3. If the machine is compromised at the memory level, the attacker has bigger problems.
4. The practical security boundary is: keys are encrypted at rest (Credential Manager) and only travel over HTTPS.

Dimension 4: UI/UX Specifications

4.1 Design Approach: Desktop-Native, Inspired by Claude.ai

> DECISION: No Figma mockup. Reference Claude.ai's visual language but design for desktop conventions.

The developer should reference the Claude.ai web interface for:

- Color palette: warm neutrals (#F5F5F0 background, #1A1A1A text, #D97706 accent/brand orange)
- Chat bubble styling: left-aligned assistant, right-aligned user, with rounded corners
- Sidebar layout proportions

But adapt for desktop conventions:

- Use native window chrome (title bar, minimize/maximize/close)
- Support window snapping (Aero Snap)
- Respect Windows system font size settings
- Use system file dialogs for uploads

4.2 Theme System

> DECISION: Light and Dark mode with auto-detection from Windows system setting.

Implementation:

1. Check Windows registry or use darkdetect library to detect system theme preference.
2. Provide manual override in Settings: Auto / Light / Dark.
3. Use Qt stylesheet (QSS) for theming. Define all colors as variables in a theme dict.

Color scheme:

Element	Light Mode	Dark Mode
Background	#FFFFFF	#1E1E1E
Sidebar	#F5F5F0	#252525
User Bubble	#E8F0FE	#2A3A4A

Assistant Bubble	#F8F8F8	#2D2D2D
Accent/Brand	#D97706 (Claude orange)	#F59E0B
Code Block BG	#F5F5F5	#1A1A1A
Text Primary	#1A1A1A	#E5E5E5

4.3 Markdown Rendering: QWebEngineView

> DECISION: Use QWebEngineView for chat message rendering.

Reasons:

1. Full HTML/CSS/JS support means perfect Markdown rendering, including code blocks with syntax highlighting, tables, LaTeX math (via KaTeX), and Mermaid diagrams.
2. Streaming text can be appended via JavaScript injection (page.runJavaScript) without re-rendering the entire page.
3. The Chromium dependency adds approximately 80-100MB to the distribution, but this is acceptable for a desktop application.

Rendering pipeline:

User/Assistant message (Markdown string)

-> python-markdown + pymdown-extensions (convert to HTML)

-> Pygments (syntax highlighting for code blocks)

-> Inject into QWebEngineView template

Alternative fallback: If QWebEngineView proves too heavy during development, use QTextBrowser with basic HTML rendering as an interim solution. But QWebEngineView is the target.

4.4 Fonts

> DECISION: System fonts with fallback chain.

UI Text: "Segoe UI", "Microsoft YaHei UI", sans-serif (Segoe UI is the Windows system font; YaHei handles Chinese characters)

Code Blocks: "Cascadia Code", "JetBrains Mono", "Consolas", monospace (Cascadia Code ships with Windows Terminal and is increasingly standard on Windows 11; Consolas is the

universal fallback on Windows 10)

Do NOT bundle fonts. Use whatever the user's system provides. The fallback chain ensures good rendering on both Windows 10 and 11.

4.5 Keyboard Shortcuts

> DECISION: Follow Claude.ai conventions where applicable.

Shortcut	Action
Ctrl + Enter	Send message
Shift + Enter	New line in input (default behavior)
Ctrl + N	New conversation in current project
Ctrl + Shift + N	New project
Ctrl + P	Quick switch project (search popup)
Ctrl + F	Search in current conversation
Ctrl + Shift + F	Search across all conversations
Ctrl + ,	Open settings
Ctrl + L	Focus input box
Ctrl + [/]	Toggle sidebar / right panel
Escape	Cancel streaming response

Dimension 5: Feature Boundaries & MVP Scope

5.1 Document Processing Decisions

5.1.1 PDF Library: PyMuPDF (fitz)

> DECISION: Use PyMuPDF as primary, with pdfplumber as optional fallback.

PyMuPDF is significantly faster (10-50x) and handles most PDFs well. Its AGPL license is acceptable because we are not distributing PyMuPDF as a library; we are using it as a tool in an end-user application. If there are specific PDFs where PyMuPDF produces poor output, implement a fallback to pdfplumber for that specific document.

5.1.2 OCR Support: NOT in MVP

> DECISION: No OCR in Phase 1-3. Add as Phase 4 stretch goal.

Scanned PDF handling (Tesseract OCR) is a significant complexity addition. For the MVP, if PyMuPDF cannot extract text from a PDF (i.e., it's a pure image PDF), display a warning to the user: "This PDF appears to be a scanned image. Text extraction is not available. You can still include it as an image attachment in individual messages."

If OCR is added later, use pytesseract + pdf2image. This requires installing Tesseract binary separately, which should be documented.

5.1.3 Image Storage

> DECISION: Store original files on disk. Generate base64 only at API call time.

When the user attaches an image to a message:

1. Save the original image to ~/ClaudeStation/projects/{project_id}/attachments/{uuid}.{ext}
2. Store the file path in the messages.attachments_json field
3. At API call time, read the file, base64-encode it, and include in the content block
4. Do NOT store base64 in the database (it bloats SQLite unnecessarily)

5.2 Context Compression: Keyword Matching Only

> DECISION: Use simple keyword/TF-IDF matching for document relevance filtering. NO local embedding model.

Rationale: A local embedding model (sentence-transformers) adds 200MB+ to the distribution, requires PyTorch, and introduces GPU/CPU compatibility issues. The ROI is not justified for the MVP.

Implementation:

5. When building the API request, extract keywords from the last 3 user messages using simple tokenization + stopword removal.
6. Score each project document by the number of keyword matches (TF-IDF style).
7. Include the top N documents by score (where N is configurable, default: all documents if total tokens < 60% of context window, otherwise top-scoring subset).
8. If ALL documents fit within the cache budget, always include all of them (cached content is cheap to read).

> **Important:** Document pruning should only activate when the total project documents exceed the context budget. If the documents are cached and total under 100K tokens, include everything. The 10% cache read price makes inclusion cheap.

5.3 Token Counting Strategy

> DECISION: Dual approach - pre-estimate with tiktoken, post-verify with API response.

Implementation:

Pre-send estimation: Use tiktoken with the cl100k_base encoding to estimate token count before sending. This gives the user a cost preview (e.g., "~12,450 input tokens, estimated cost: $0.037"). The estimate may be off by 5-10% but is sufficient for user guidance.

Post-send tracking: Use the actual usage fields from the API response (input_tokens, output_tokens, cache_creation_input_tokens, cache_read_input_tokens) for accurate cost tracking and dashboard display.

Token count display: Show the estimate before sending (in the bottom bar), and update to actual counts after the response completes (in the message metadata).

5.4 Export Functionality

> DECISION: Markdown and JSON export only. No PDF/DOCX export in MVP.

Export formats:

Markdown (.md): Clean, human-readable format. Each message is formatted as '## User / ## Assistant' with the content below. Includes a YAML frontmatter header with conversation metadata (title, model, date range).

JSON (.json): Machine-readable format containing the full conversation data including all message metadata, token counts, costs, model used, and timestamps. This format can be re-imported into the application.

Token statistics: YES, include in both formats. Markdown gets a summary section at the end (total tokens, total cost, model breakdown). JSON includes per-message token data.

Appendix A: Updated requirements.txt

Based on all decisions above, here is the final dependency list:

```
# Core
anthropic>=0.40.0          # Anthropic Python SDK (Messages API + streaming)
PySide6>=6.6.0             # Qt 6 GUI framework (LGPL)
PySide6-Addons>=6.6.0      # QtWebEngineWidgets for markdown rendering

# Document Processing
PyMuPDF>=1.24.0            # PDF text extraction (fast)
pdfplumber>=0.11.0         # PDF fallback extraction (accurate)
python-docx>=1.1.0         # DOCX text extraction
openpyxl>=3.1.0            # XLSX text extraction
chardet>=5.0               # File encoding detection

# Token & Cost
tiktoken>=0.7.0            # Token counting (pre-estimation)

# Security
keyring>=25.0              # Windows Credential Manager integration

# UI Rendering
markdown>=3.5              # Markdown to HTML conversion
pymdown-extensions>=10.0   # Extended markdown (tables, task lists, etc.)
Pygments>=2.17             # Code syntax highlighting

# Image
Pillow>=10.0               # Image resizing/conversion before API calls

# Utilities
darkdetect>=0.8.0          # Windows dark/light theme detection
httpx[socks]>=0.27.0       # HTTP client with SOCKS proxy support
```

```
# Development
ruff>=0.4.0              # Linting + formatting
mypy>=1.10               # Type checking

# REMOVED from original PRD:
# qasync - not needed, using QThread + asyncio.new_event_loop()
```

Appendix B: Revised Development Phase Order

Based on the developer's suggested approach (which is good), here is the refined sequence:

Phase 0: Environment & API Validation (Day 1-2)

6. Set up Python 3.12 venv with all dependencies from Appendix A
7. Write a minimal script that: (a) sends a message to claude-sonnet-4-5-20250929 with streaming, (b) uses cache_control on a system prompt, (c) prints cache_creation_input_tokens and cache_read_input_tokens from two successive calls to verify caching works
8. Test proxy connectivity if needed (httpx.Client with proxy parameter)

Phase 1: Data Layer + Skeleton UI (Week 1)

5. Create SQLite schema (projects, documents, conversations, messages, api_keys tables)
6. Implement ProjectManager and ConversationManager CRUD
7. Build main window with three-panel layout (empty panels, just the shell)
8. Implement API key management with keyring (Settings dialog)

Phase 2: Core Chat Flow (Week 2-3)

4. Implement ClaudeAPIClient with streaming (QThread + asyncio)
5. Build ContextBuilder (system prompt + documents + history assembly)
6. Implement chat widget with QWebEngineView markdown rendering
7. Wire up: create project -> new conversation -> send message -> display streaming response -> save to DB
8. Add model selector dropdown

Phase 3: Projects Feature (Week 3-4)

4. Document upload and text extraction (PDF, DOCX, TXT, code)
5. System prompt editor per project
6. Prompt caching integration (cache_control on system + docs block)
7. Token usage tracking with cost calculation
8. Multiple conversations per project in sidebar

Phase 4: Optimization & Polish (Week 5-6)

5. Context compression (sliding window + Haiku summarization)
6. Cache hit rate monitoring dashboard
7. Dark/light theme with system auto-detection
8. Keyboard shortcuts
9. Extended thinking toggle
10. Image attachments
11. Conversation search and export (MD/JSON)
12. Error recovery, retry UI, proxy configuration

Summary of All Decisions

Question	Decision
Model versions	ALL CORRECT. Claude 4.5 series is current. Developer needs to update knowledge.
Prompt Caching API	GA, no beta header needed. Just use cache_control field.
Pricing	Correct as stated. Opus 4.5: $5/$25, Sonnet 4.5: $3/$15, Haiku 4.5: $1/$5.
GUI Framework	PySide6 (LGPL, official Qt binding).
Python Version	3.11+ minimum, develop on 3.12.
Async Architecture	QThread + asyncio.new_event_loop() in worker thread. No qasync.
Code Standards	PEP 8 + mandatory type hints + Google docstrings. Line length 100.
Proxy Support	Built-in from Day 1. HTTP/HTTPS/SOCKS5 via httpx.

Multi API Key	Yes, support multiple profiles stored in keyring.
Backup	Manual ZIP export/import of data directory.
API Key Memory	Standard approach. No aggressive scrubbing.
Design Approach	Claude.ai-inspired, desktop-native. No Figma needed.
Theme	Light/Dark + auto-detect Windows system. QSS-based.
Markdown Renderer	QWebEngineView (Chromium-based, full HTML/JS).
Fonts	System fonts: Segoe UI + YaHei for UI, Cascadia Code for code.
Shortcuts	Claude.ai-compatible. See full table in Section 4.5.
PDF Library	PyMuPDF primary, pdfplumber fallback.
OCR	Not in MVP. Phase 4 stretch goal.
Image Storage	Original files on disk. Base64 generated at API call time.
Document Filtering	Keyword/TF-IDF only. No embedding models.
Token Counting	tiktoken pre-estimate + API response post-verify.
Export	Markdown + JSON only. Includes token/cost stats.

> **Next Step:** Developer should proceed with Phase 0 (environment setup + API validation script) immediately. All blocking questions are now resolved.