ClaudeStation 综合产品需求文档（PRD）

版本: 3.0（整合版）
日期: 2026-02-16
状态: 设计完成，待实施
分支: `fix/chat-persistence-and-cleanup`
仓库: `github.com/JustinZhu5268/CladueProjectsDesk`

文档说明

本文档是以下三份文档的**整合与升级版**：

- **文档 1**：Claude Desktop Client PRD（High-Level & Low-Level Design）
- **文档 2**：PRD Clarification Response（技术栈决策与澄清）
- **文档 3**：Token 优化与对话压缩 PRD v2.0

整合过程中吸纳了外部专家审核意见，解决了三份文档之间的**逻辑冲突**，并在工程落地层面做了增强。所有冲突解决和变更决策均有标注。

第一部分：产品概述

1.1 产品名称与定位

ClaudeStation — 一个连接 Anthropic Claude API 的 Windows 桌面客户端，提供与 Claude.ai Projects 等价的功能集，同时具备网页版不具备的 **token 经济性控制能力**。

1.2 目标用户

- 需要通过 API 管理复杂多文档项目的开发者和研究者
- 需要完全控制模型选择、token 预算和对话管理的 Power User
- 需要本地数据主权、离线文档管理 + 云端 AI 推理的用户

1.3 核心价值主张

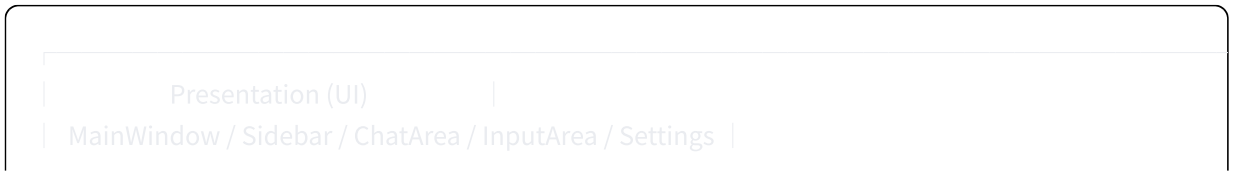
1. **项目制组织**：复刻 Claude.ai Projects，支持本地文档上传和每项目多对话
2. **模型灵活性**：支持全线 Claude 4.5+ 模型，实时切换
3. **Token 经济性**：双层 Prompt Cache + 增量式对话压缩 + Compaction API 兜底，相比网页版节省 60-73% 输入成本
4. **数据主权**：所有项目数据本地存储，仅 API 调用离开本机

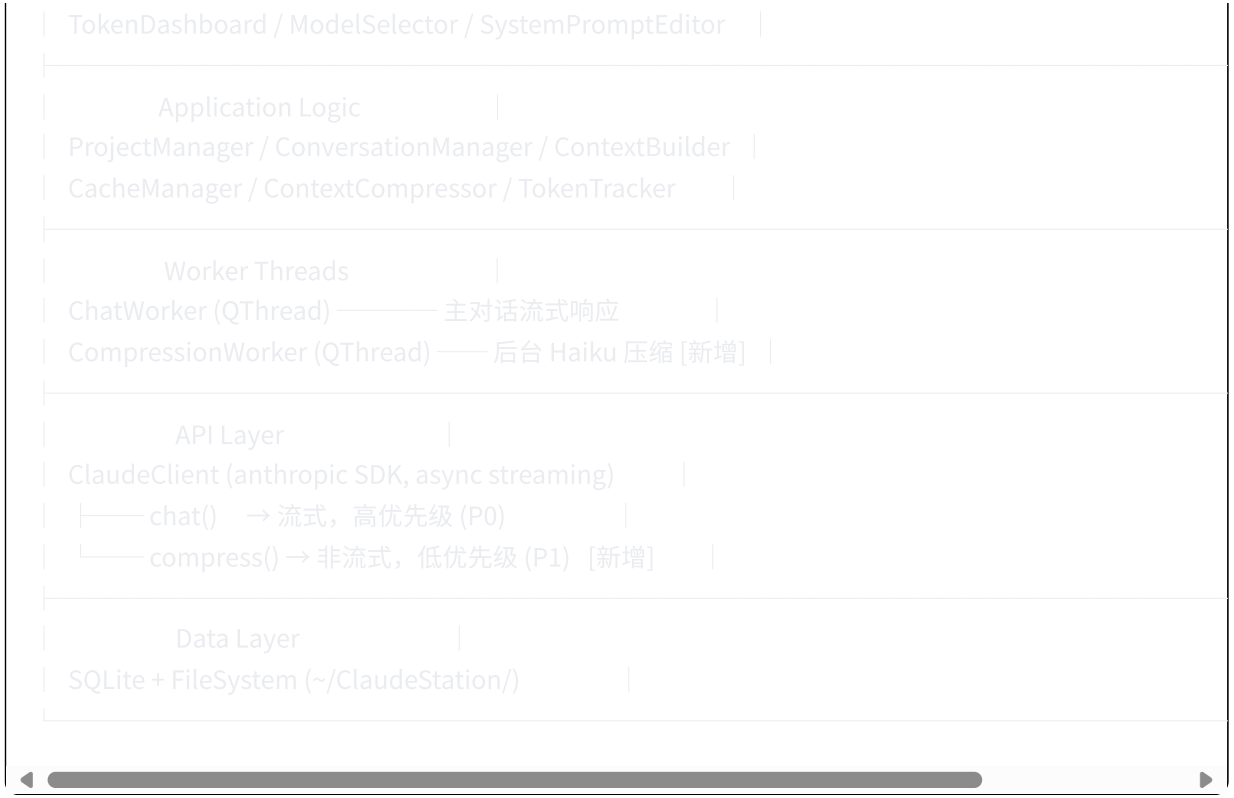
第二部分：技术栈与架构决策

2.1 技术栈（最终决策）

组件	决策	理由
GUI 框架	PySide6 ≥6.6.0	LGPL 许可，官方 Qt for Python
Python 版本	3.11+ (开发用 3.12)	性能提升 10-60%，asyncio 改进
异步架构	QThread + asyncio.new_event_loop()	~~qasync~~ 已移除，避免 UI 卡顿
Markdown 渲染	QWebEngineView	支持代码高亮、表格、LaTeX
数据库	SQLite	单文件，零部署
API SDK	anthropic ≥0.40.0	官方 Python SDK
代码规范	PEP 8 + 类型标注 + Google docstring	ruff 检查，行长 100

2.2 系统架构（四层 + 压缩 Worker）





> **【专家建议采纳】架构新增 CompressionWorker**
> 文档 1 原始架构只有 ChatWorker。专家指出压缩如果在主聊天线程中执行，会增加用户发送消息时的延迟。本整合版新增 CompressionWorker 作为独立 QThread，在 `response_complete` 信号后静默执行 Haiku 压缩。

> **【专家建议采纳】API 客户端拆分**
> ClaudeClient 拆分为 `chat()`（流式，用户选择的模型）和 `compress()`（非流式，强制使用 Haiku）两个方法。当触发 Anthropic 速率限制时，优先保证 `chat()` 请求，暂停 `compress()` 请求。

2.3 目录结构

```
claude_station/
├—— main.py              # 入口，初始化 MainWindow 和 DB
├—— config.py            # 全局配置（路径、常量、压缩参数）
├—— requirements.txt     # 依赖
```

```
│
├── api/
│   ├── claude_client.py      # Anthropic SDK 封装（chat + compress 方法）
│   ├── models.py             # 模型定义、定价、上下文窗口
│   └── error_handler.py      # 重试逻辑、限流、错误反馈
│
├── core/
│   ├── database_manager.py   # SQLite 连接、迁移
│   ├── project_manager.py    # 项目 CRUD
│   ├── conversation_manager.py # 对话 CRUD + compress_history() [新增]
│   ├── context_builder.py    # 四层上下文构建 [重写]
│   ├── context_compressor.py # 增量式滚动摘要算法 [重写]
│   ├── cache_manager.py      # Cache 命中率监控
│   ├── document_processor.py # PDF/DOCX/TXT 文本提取
│   ├── token_counter.py      # tiktoken 预估
│   ├── token_tracker.py      # API 响应 token 追踪与成本计算
│   └── signals.py            # 全局 Qt 信号
│
├── data/
│   ├── schema.sql            # 表定义（含压缩字段）[已更新]
│   └── file_storage.py       # 文档文件管理
│
├── ui/
│   ├── main_window.py        # 主窗口（三面板布局）
│   ├── chat_area.py          # 聊天区（QWebEngineView）
│   ├── input_area.py         # 输入区
│   ├── sidebar.py            # 左侧栏（项目/对话列表）
│   ├── settings_dialog.py    # 设置（含 Cache TTL、压缩参数）[扩展]
│   ├── token_dashboard.py    # Token 仪表盘（含缓存命中率）[扩展]
│   ├── model_selector.py     # 模型下拉（含成本提示）
│   └── styles.py             # 主题 QSS
│
└── utils/
    ├── markdown_renderer.py  # MD→HTML + js_safe() 安全编码
    └── export.py             # 对话导出（MD/JSON）
```

第三部分：数据库设计

3.1 完整 Schema

> **【冲突解决】** 文档 1 的 `conversations` 表缺少压缩状态字段。本版本合并了文档 3 建议的字段，并增加了 `projects` 表的压缩配置。

projects 表

列名	类型	可空	说明
id	TEXT PK	NO	UUID v4
name	TEXT	NO	项目显示名称
system_prompt	TEXT	YES	自定义系统提示词
default_model	TEXT	NO	默认模型 ID
created_at	DATETIME	NO	创建时间
updated_at	DATETIME	NO	最后修改时间
settings_json	TEXT	YES	JSON：effort、thinking budget、**cache_ttl**、**compress_after_turns** 等

documents 表

列名	类型	可空	说明
id	TEXT PK	NO	UUID v4
project_id	TEXT FK	NO	引用 projects.id
filename	TEXT	NO	原始文件名
file_path	TEXT	NO	存储相对路径
extracted_text	TEXT	YES	提取的纯文本
token_count	INTEGER	YES	提取文本的 token 数
file_type	TEXT	NO	MIME 类型或扩展名
created_at	DATETIME	NO	上传时间

conversations 表（已扩展）

列名	类型	可空	说明
id	TEXT PK	NO	UUID v4
project_id	TEXT FK	NO	引用 projects.id
title	TEXT	NO	自动或用户设定的标题
model_override	TEXT	YES	覆盖项目默认模型
created_at	DATETIME	NO	创建时间
updated_at	DATETIME	NO	最后消息时间
is_archived	BOOLEAN	NO	归档标志
rolling_summary	**TEXT**	YES	**累积的对话摘要** [新增]
last_compressed_msg_id	**TEXT**	YES	**摘要覆盖到哪条消息的 ID** [新增]
summary_token_count	**INTEGER**	YES	**摘要的 token 数** [新增]

messages 表

列名	类型	可空	说明
id	TEXT PK	NO	UUID v4
conversation_id	TEXT FK	NO	引用 conversations.id
role	TEXT	NO	"user" 或 "assistant"
content	TEXT	NO	消息文本
thinking_content	TEXT	YES	Extended Thinking 输出
attachments_json	TEXT	YES	JSON：附件图片路径
model_used	TEXT	YES	此条回复使用的模型
input_tokens	INTEGER	YES	输入 tokens
output_tokens	INTEGER	YES	输出 tokens
cache_read_tokens	INTEGER	YES	缓存读取 tokens
cache_creation_tokens	INTEGER	YES	缓存写入 tokens
cost_usd	REAL	YES	计算的 USD 成本
created_at	DATETIME	NO	时间戳

api_keys 表

列名	类型	说明
id	TEXT PK	UUID
label	TEXT	显示名（如 "Personal", "Work"）
key_ref	TEXT	keyring 存储引用 ID（非密钥本身）
is_default	BOOLEAN	是否为默认密钥
created_at	DATETIME	创建时间

第四部分：Token 优化与对话压缩系统（核心创新）

> 本部分是三份文档中争议最多、也是 ClaudeStation 相比网页版**最大的差异化竞争力**所在。以下是经过纠错和专家审核后的最终设计。

4.1 前期讨论中的关键纠正

纠正 1：1 小时缓存的写入成本是 2.0x，不是 1.25x

之前的讨论混淆了两种 TTL 的定价。官方实际定价：

TTL	写入成本乘数	读取成本乘数	API 语法
5 分钟（默认）	1.25x	0.1x	`{"type": "ephemeral"}`
1 小时	**2.0x**	0.1x	`{"type": "ephemeral", "ttl": "1h"}`

决策：默认使用 5 分钟 TTL。在 Settings 中提供 1 小时选项，附说明："如果你经常中断超过 5 分钟，开启此选项可节省成本"。

纠正 2：放弃心跳保活机制

2 次心跳成本约 20% vs 缓存重建 25%，仅省 5%，但实现复杂度高且有竞态风险。**不值得实现**。

纠正 3：Anthropic 已提供官方 Compaction API

2026 年 1 月发布的 `compact-2026-01-12` beta API 可在服务端自动压缩。但触发时机较晚（接近 200K 极限），无法替代客户端主动压缩。**采用"客户端主动压缩 + Compaction API 兜底"的混合策略。**

纠正 4：压缩成本必须用 Haiku

之前的分层压缩方案用主模型做压缩，未考虑输出价格是输入价格的 5 倍。正确做法是**强制使用 Haiku（$1/$5）做压缩**，单次摩擦仅 ~$0.006。

4.2 核心设计原则

1. **缓存前缀最大化**：尽可能多的稳定内容放入缓存前缀，享受 0.1x 读取价格
2. **未缓存区域最小化**：通过压缩历史对话，减少每轮实际发送的未缓存 token 数
3. **压缩成本最低化**：用 Haiku 做压缩，增量式而非批量
4. **精度保持最大化**：最近 N 轮对话保留全文，只压缩较老的历史

4.3 四层上下文架构

> **【冲突解决】** 文档 1 描述的是简单的 `System + Docs + History` 三层结构。本版本升级为四层结构，支持多重 Cache Breakpoint。

每次 API 调用的输入结构分为四层：

层次	内容	缓存状态	变化频率	成本
缓存层 1	System Prompt + 项目文档	第一个 cache breakpoint	极少变化	写入 1.25x，读取 0.1x
缓存层 2	对话摘要 (Rolling Summary)	第二个 cache breakpoint	每 ~10 轮更新	写入 1.25x，读取 0.1x
未缓存层	最近 8-10 轮完整对话	无缓存	每轮变化	基础输入价 1.0x
当前消息	用户新消息 + 附件	无缓存	每轮变化	基础输入价 1.0x

这是本方案的核心创新：将对话摘要作为第二个 cache breakpoint。 因为摘要每 10 轮才更新一次，在这 10 轮内它也被缓存读取（只付 0.1x）。而旧方案的 30K+ 历史 tokens 全部以 1.0x 价格发送。

> **【专家建议采纳】1024 Token 缓存门槛**
> Anthropic API 要求至少 1024 tokens 的 Block 才能被缓存。如果 Rolling Summary 刚开始生成、只有几百 tokens，它无法作为有效的 cache breakpoint。在此阶段，摘要以普通文本（1.0x）发送。只有积累到 >1024 tokens 后，才标记 `cache_control`。

> **【专家建议采纳】缓存链式失效**
> Anthropic 的 Prompt Caching 基于**前缀匹配**。如果 Layer 1 发生任何变化（即使 1 个字节），不仅 Layer 1 的 cache 失效，**紧随其后的 Layer 2 也会失效**（即使摘要本身没变）。因此：(a) 在对话中途修改系统提示词或文档时，状态栏提示"修改将导致缓存失效，下一次对话成本将短暂上升"；(b) 代码中确保 System Prompt + Docs 的字节级一致性。

4.4 增量式滚动摘要算法

4.4.1 触发条件

当未压缩的对话轮次超过 N 轮（默认 N=10）时触发压缩。

4.4.2 压缩方式

取最早的 K 轮（默认 K=5）对话，调用 Haiku 生成摘要，追加到 Rolling Summary。

4.4.3 关键特性

每次只压缩 5 轮（而非全部历史），压缩成本固定且可预测。无论对话多长，每次压缩的输入量始终约 3K tokens。

4.4.4 算法流程（异步预压缩模式）

> **【专家建议采纳】异步预压缩**
> 原设计暗示在用户发送第 11 条消息时才触发压缩。专家指出这会导致 Haiku 压缩耗时 + Sonnet 回复耗时 的叠加延迟。本版本改为"响应后触发"模式。

```
用户发送第 1-10 条消息 → 正常处理，全文发送
                ↓
用户收到第 10 条回复（满一批）
                ↓
CompressionWorker 后台启动 → 取第 1-5 轮，调用 Haiku 压缩
                → 将摘要存入 DB (conversations.rolling_summary)
                → 标记 last_compressed_msg_id
                ↓
用户发送第 11 条消息 → 从 DB 读取已准备好的 Summary
            → 构建: [Layer1] + [Summary] + [第 6-10 轮全文] + [第 11 条]
            → 无额外等待
                ↓
用户发送第 16-20 条  → 正常使用 Summary + 近期全文
用户收到第 20 条回复  → CompressionWorker 再次启动
            → 取第 6-10 轮，追加压缩到 Summary
                ↓
循环...
```

4.4.5 压缩提示词模板

> **【专家建议采纳】代码块保护**
> 专家指出 Haiku 生成的摘要可能将代码块概括为自然语言描述（如"用户提供了一个斐波那契函数"），导致后续 Claude 无法执行代码重构。本版本在压缩提示词中明确要求保留代码签名和核心逻辑。

> **【专家建议评估】Haiku 上下文盲区**
> 专家提出悖论：是否将项目文档传给 Haiku？传入则成本飙升，不传则可能丢失领域术语。**决策：不传项目文档。** 压缩任务是语言学层面的概括，不需要领域知识。压缩提示词中包含最小化的指导（项目名称和"保留专业术语原文"的指令），但不包含完整文档。这将单次压缩成本控制在 ~$0.006。

发送给 Haiku 的压缩请求结构：

```python
# system prompt（极简，~100 tokens）
system = "You are a conversation summarizer for project '{project_name}'. "
    "Output ONLY the summary in the same language as the conversation. "
    "No preamble, no explanation."

# user prompt（~200 tokens 指令 + ~3K tokens 对话内容）
user = """请将以下对话压缩为简洁摘要。规则：
1. 保留所有关键决策和结论
2. 保留代码片段的函数签名和核心逻辑（不要只用自然语言概括代码）
3. 保留数据点、技术细节、专业术语原文
4. 保留用户偏好和约束条件
5. 删除客套、闲聊、重复内容
6. 摘要长度控制在 500 tokens 以内

现有摘要:
{existing_summary}

新对话内容:
{conversation_turns}"""
```

4.4.6 摘要质量保障

1. **原始消息不删除**：数据库保留全部原始消息，仅在构建 API 请求时使用摘要替代。导出时输出完整历史。

2. **摘要长度控制**：每次压缩产生的摘要不超过 500 tokens，累积 5 次后约 2000-2500 tokens。
3. **摘要增长监控**：如果 Rolling Summary 超过 3000 tokens，对摘要本身再做一次压缩（"摘要的摘要"）。
4. **重置摘要按钮**：在 UI 提供"清除摘要 / 重建上下文"按钮，清空 `rolling_summary`，强制下一次请求发送全量历史。

> **【专家建议采纳】重置按钮**
> 如果摘要出现幻觉或压缩错了关键信息，用户需要一种"后悔药"。

4.5 Compaction API 兜底集成

客户端压缩在对话中期主动优化成本，但如果用户的单次消息非常长（如粘贴大量代码），仍可能接近上下文极限。Compaction API 作为安全网：

```python
# 在 ClaudeClient.chat() 的 API 调用参数中添加
kwargs["betas"] = ["compact-2026-01-12"]
kwargs["context_management"] = {
  "edits": [{
    "type": "compact_20260112",
    "trigger": {"type": "input_tokens", "value": 160000}  # 200K 的 80%
  }]
}
```

客户端压缩在 ~60K 时就开始工作，所以 Compaction API 通常不会触发。它只在异常情况下生效。

4.6 ContextBuilder 实现规格（重写）

> **【冲突解决】** 文档 1 Section 8.2 描述的 3 层构建逻辑已被本节替代。

```python
```

```
def build_payload(self, conversation, user_message):
    project = self.project_manager.get(conversation.project_id)
    docs_text = self.doc_processor.get_project_context(project.id)
    settings = json.loads(project.settings_json or '{}')
    cache_ttl = settings.get("cache_ttl", "5m")

    # ── Layer 1: System + Docs (Cache Breakpoint 1) ──
    cache_config = {"type": "ephemeral"}
    if cache_ttl == "1h":
        cache_config["ttl"] = "1h"

    system_content = [{
        "type": "text",
        "text": project.system_prompt + "\n\n" + docs_text,
        "cache_control": cache_config,
    }]

    # ── Layer 2: Rolling Summary (Cache Breakpoint 2) ──
    if conversation.rolling_summary:
        summary_block = {
            "type": "text",
            "text": f"<conversation_summary>\n{conversation.rolling_summary}\n</conversation_summary
        }
        # 仅当摘要超过 1024 tokens 时才标记为缓存断点
        if (conversation.summary_token_count or 0) >= 1024:
            summary_block["cache_control"] = cache_config
        system_content.append(summary_block)

    # ── Layer 3: Recent Messages (未缓存) ──
    all_messages = self.conv_mgr.get_messages(conversation.id)
    if conversation.last_compressed_msg_id:
        # 只取未压缩的消息
        messages = messages_after(all_messages, conversation.last_compressed_msg_id)
    else:
        messages = all_messages
```

```
    # Token 预算约束
    messages = self._fit_to_budget(messages, model_context_window=200000)

    # ── Layer 4: Current User Message ──
    messages.append({"role": "user", "content": user_message})

    return {"system": system_content, "messages": messages}
```

4.7 成本预估逻辑修正

> **【冲突解决：关键】** 文档 2 Section 5.3 规定"发送前使用 tiktoken 估算全量 token"。文档 3 引入压缩后，实际发送量远小于全量历史。如果不修正，用户看到的预估成本会虚高数倍，打击使用意愿。

修正方案： `CostEstimator` 必须复用 `ContextBuilder` 的构建逻辑：

```python
def estimate_cost(self, conversation, user_message, model):
    payload = self.context_builder.build_payload(conversation, user_message)
    system_tokens = count_tokens(payload["system"])
    message_tokens = count_tokens(payload["messages"])

    # 预判缓存状态
    if self._likely_cache_hit(conversation):
        input_cost = system_tokens * model.cache_read_price + message_tokens * model.input_price
    else:
        input_cost = (system_tokens + message_tokens) * model.input_price

    return {
        "estimated_input_tokens": system_tokens + message_tokens,
        "estimated_input_cost": input_cost,
        "cached_tokens": system_tokens if self._likely_cache_hit(conversation) else 0,
    }
```

UI 显示："预计 $0.05 (含缓存节省 85%)"，而不是虚报 $0.75。

4.8 压缩失败容错

> **【专家建议采纳】降级处理**
> 如果 Haiku 压缩服务不可用（网络问题、余额不足、限流），不能阻断用户主流程。

```python
async def compress_history(self, conversation_id):
    try:
        summary = await self.client.compress(oldest_turns, existing_summary)
        self.db.update_summary(conversation_id, summary)
    except Exception as e:
        log.warning(f"Compression failed: {e}. Will use full history next turn.")
        # 不修改 summary，下次 ContextBuilder 会发送全量历史
        # 状态栏显示黄色警告："压缩服务不可用，正在使用全量上下文"
```

4.9 Cache TTL 选择决策树

基于 50K tokens 项目文档、Sonnet 定价：

场景	5 分钟 TTL	1 小时 TTL	推荐
持续对话，从不超过 5 分钟空闲	初始 $0.19，续期 $0.015/轮	初始 $0.30	5 分钟（更便宜）
偶尔超过 5 分钟 (1 次/小时)	$0.375	$0.30	1 小时（更便宜）
经常超过 5 分钟 (3 次/小时)	$0.75	$0.30	1 小时（明显更便宜）

4.10 可配置参数汇总

参数	默认值	范围	位置
COMPRESS_AFTER_TURNS	10	5-30	config.py + Settings UI

参数	默认值	范围	位置
COMPRESS_BATCH_SIZE	5	3-10	config.py
MAX_SUMMARY_TOKENS	500	200-1000	config.py
SUMMARY_RECOMPRESS_THRESHOLD	3000	2000-5000	config.py
CACHE_TTL	"5m"	"5m" / "1h"	Settings UI
COMPACTION_TRIGGER	160000	100000-180000	config.py
RECENT_TURNS_KEPT	10	5-20	config.py

Settings UI 提供两个预设，而非直接暴露数字：

> **【专家建议采纳】压缩策略预设**

- **标准模式**（默认）：N=10, K=5 — 平衡成本与上下文质量
- **保守模式**：N=20, K=5 — 保留更多完整上下文，适合代码 debug 场景

4.11 成本对比分析

以 Sonnet 4.5（$3/$15）、50K 项目文档、50 轮对话为例：

指标	方案 A: 当前简单截断	方案 B: 增量摘要+双层缓存	方案 C: 网页版 Claude.ai
第 50 轮发送量	50K(缓存)+45K(历史)	50K(缓存)+2K(摘要缓存)+9K(近期)	无控制
缓存读取量	50K @ 0.1x	52K @ 0.1x	无缓存
未缓存量	45K @ 1.0x	9K @ 1.0x	全部 @ 1.0x
第 50 轮输入成本	$0.150	$0.043	无法计算
50 轮累计总输入成本	~$3.60	~$1.35	~$5.00+
节省比例 (vs 方案 A)	基线	**62% 节省**	-
节省比例 (vs 网页版)	28%	**73% 节省**	基线

第五部分：Prompt Caching 实现细节

5.1 缓存结构

每次 API 调用的请求结构：

```
python
```

```
client.messages.create(
  model=selected_model_id,
  max_tokens=8192,
  system=[
    # Cache Breakpoint 1: System + Docs
    {"type": "text",
     "text": system_prompt + "\n\n" + all_docs_text,
     "cache_control": {"type": "ephemeral"}},
    # Cache Breakpoint 2: Rolling Summary (条件性)
    {"type": "text",
     "text": f"<conversation_summary>{summary}</conversation_summary>",
     "cache_control": {"type": "ephemeral"}}, # 仅 >1024 tokens 时
  ],
  messages=recent_conversation_history + [{"role": "user", "content": user_input}],
  stream=True,
  # Compaction API 兜底
  betas=["compact-2026-01-12"],
  context_management={"edits": [{"type": "compact_20260112", "trigger": {"type": "input_tokens", "v
)
```

5.2 缓存优化规则

1. **永远不要在同一会话的轮次之间修改系统提示词内容**。即使空格变化也会使缓存失效。
2. 用户中途上传/删除文档时，Layer 1 缓存失效重建（预期行为），**Layer 2 也会因前缀变化而失效**。状态栏提示用户。
3. 用户切换模型时，所有缓存失效。状态栏显示重建成本估算。
4. 监控每个 API 响应的 `cache_read_input_tokens`。如果意外降为 0，记录警告日志。

5.3 CacheManager

- 追踪每次 API 响应的 `cache_creation_input_tokens` 和 `cache_read_input_tokens`
- 计算缓存命中率 = `cache_read / (cache_read + cache_creation)`
- 如果命中率降至 50% 以下，在 Token Dashboard 中显示警告

- 提供会话级缓存节省报告

第六部分：模型定价与配置

6.1 模型定价表

模型	输入 $/MTok	输出 $/MTok	Cache 写入 (5min)	Cache 写入 (1h)	Cache 读取	上下文	模型 ID
Opus 4.6	$5.00	$25.00	$6.25	$10.00	$0.50	200K	claude-opus-4-6
Opus 4.5	$5.00	$25.00	$6.25	$10.00	$0.50	200K	claude-opus-4-5-20251101
Sonnet 4.5	$3.00	$15.00	$3.75	$6.00	$0.30	200K	claude-sonnet-4-5-20250929
Haiku 4.5	$1.00	$5.00	$1.25	$2.00	$0.10	200K	claude-haiku-4-5-20251001

> Sonnet 4.5 输入超过 200K tokens 时适用长上下文定价（2x），本应用通过压缩系统避免触及此阈值。

6.2 成本计算公式

```python
```

```
cost = (
  input_tokens * input_price / 1_000_000
  + output_tokens * output_price / 1_000_000
  + cache_creation_tokens * input_price * cache_write_multiplier / 1_000_000
  + cache_read_tokens * input_price * 0.1 / 1_000_000
)
# cache_write_multiplier: 1.25 (5min) 或 2.0 (1h)
```

6.3 模型切换成本提示

当用户切换模型时，状态栏静默显示（不弹窗）：

```
"切换到 Opus 4.6 — 项目文档 50K tokens，首次 cache 写入成本约 $0.31"
```

第七部分：核心模块规格

7.1 ClaudeAPIClient (`api/claude_client.py`)

方法拆分

python

```python
class ClaudeClient:
  async def chat(self, messages, system, model, stream=True, **kwargs):
    """主对话方法。流式，高优先级。使用用户选择的模型。"""
    # 包含 Compaction API 兜底参数
    # 包含 cache_control 注入
    # 指数退避重试 (3 次，1s/2s/4s)

  async def compress(self, turns, existing_summary, project_name):
    """压缩方法。非流式，低优先级。强制使用 Haiku。"""
    # 不包含项目文档上下文
    # 使用精简版 system prompt
    # 无重试（失败则跳过）
```

限流优先级

当触发 Anthropic RPM/TPM 限制时：

1. **P0**: `chat()` 请求 — 立即重试
2. **P1**: `compress()` 请求 — 暂停，等限流窗口过后再执行

7.2 ContextCompressor (`core/context_compressor.py`)

```python
```

```python
class ContextCompressor:
    def should_compress(self, conversation) -> bool:
        """判断是否需要压缩"""
        uncompressed = self._get_uncompressed_messages(conversation)
        threshold = conversation.compress_after_turns or config.COMPRESS_AFTER_TURNS
        return len(uncompressed) > threshold * 2  # 每轮 2 条消息 (user + assistant)

    async def compress(self, conversation) -> str:
        """执行增量压缩，返回新的摘要"""
        uncompressed = self._get_uncompressed_messages(conversation)
        oldest_batch = uncompressed[:config.COMPRESS_BATCH_SIZE * 2]

        summary = await self.client.compress(
            turns=oldest_batch,
            existing_summary=conversation.rolling_summary or "",
            project_name=conversation.project.name,
        )

        # 更新数据库
        conversation.rolling_summary = summary
        conversation.last_compressed_msg_id = oldest_batch[-1].id
        conversation.summary_token_count = count_tokens(summary)
        self.db.save(conversation)

        # 如果摘要太长，压缩摘要本身
        if conversation.summary_token_count > config.SUMMARY_RECOMPRESS_THRESHOLD:
            await self._recompress_summary(conversation)

        return summary
```

7.3 TokenTracker (`core/token_tracker.py`)

每次 API 响应后记录：

- `input_tokens`, `output_tokens`, `cache_read_tokens`, `cache_creation_tokens`

- 计算 `cost_usd` 并存入 `messages` 表
- 更新 Token Dashboard 中的累计统计和命中率

7.4 DocumentProcessor (`core/document_processor.py`)

格式	库	备注
PDF	PyMuPDF (fitz)	主力，pdfplumber 备选
DOCX	python-docx	提取段落、表格
TXT/MD/CSV	内置 open()	chardet 编码检测
代码文件	内置 open()	语法感知截断
XLSX	openpyxl	转为 CSV 格式文本
图片	base64 + PIL	Vision 内容块，不提取文本

文档仅提取一次，存入 `documents.extracted_text`。后续对话直接使用缓存的文本。

第八部分：UI 布局与交互

8.1 主窗口布局（三面板）

- **左侧栏**（240px，可折叠）：项目列表 + 选中项目的对话列表
- **中央面板**（自适应）：QWebEngineView 聊天区 + 底部输入区
- **右侧面板**（280px，可折叠）：系统提示词编辑器 + 文档列表 + Token 仪表盘

8.2 Token 仪表盘（扩展）

> **【冲突解决】** 文档 2 的简单 token 统计已升级为含缓存命中率和压缩节省的完整仪表盘。

仪表盘显示内容：

- **每条消息**：输入/输出/缓存读取/缓存写入 tokens + 成本
- **会话累计**：总 tokens、总成本
- **Cache 命中率**：基于最近 10 次 API 调用，显示为百分比 + 进度条
- **压缩节省**："本次对话已压缩 X 次，节省约 Y tokens"
- **累计节省金额**：对比"无缓存/压缩的假设成本" vs "实际成本"

8.3 压缩状态反馈

【专家建议采纳】状态栏反馈

当 CompressionWorker 在后台执行时，状态栏显示：

> "正在整理历史记忆... (Haiku)" → "历史记忆已更新，节省 2,847 tokens"

压缩失败时：

> "⚠ 压缩服务不可用，正在使用全量上下文"（黄色警告）

8.4 Settings 扩展

在 Settings 对话框新增"Token 策略"选项卡：

- **Cache TTL**：下拉框（"5 分钟 (默认)" / "1 小时"），附说明文字
- **压缩模式**：下拉框（"标准" / "保守"），附说明文字
- **重置摘要**：按钮，"清除当前对话的摘要"

8.5 主题与字体

元素	亮色模式	暗色模式
背景	▢ #FFFFFF	● #1E1E1E
侧栏	▢ #F5F5F0	● #252525

元素	亮色模式	暗色模式
用户气泡	#E8F0FE	#2A3A4A
助手气泡	#F8F8F8	#2D2D2D
品牌色	#D97706	#F59E0B
代码块	#F5F5F5	#1A1A1A
文字	#1A1A1A	#E5E5E5

字体：UI 使用 "Segoe UI" / "Microsoft YaHei UI"，代码使用 "Cascadia Code" / "Consolas"。

8.6 键盘快捷键

快捷键	操作
Ctrl+Enter	发送消息
Shift+Enter	输入换行
Ctrl+N	新建对话
Ctrl+Shift+N	新建项目
Ctrl+P	快速切换项目
Ctrl+F	当前对话搜索
Ctrl+,	打开设置
Ctrl+L	聚焦输入框
Escape	取消流式响应

第九部分：安全与部署

9.1 API 密钥安全

- 通过 `keyring` 库使用 Windows Credential Manager 加密存储
- 支持多 API Key 配置文件（Personal / Work 等）
- API Key 运行时保持在内存中，不做激进的内存擦除（Python 字符串不可变，无法可靠清零）

9.2 代理支持

从 Day 1 内建 HTTP/HTTPS/SOCKS5 代理配置，通过 httpx 客户端传递：

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

9.3 数据备份

- 手动 ZIP 导出 `~/ClaudeStation/` 目录（不含 API Key）
- 单个项目可导出为 JSON Bundle（含对话、文档、设置）
- 对话导出为 Markdown 或 JSON（**含完整原始消息**，非压缩版）

第十部分：边界情况处理

场景	处理方式
对话不足 10 轮	不触发压缩，全文发送
Haiku 压缩调用失败	记录日志，跳过压缩，以全文继续发送

场景	处理方式
用户修改系统提示词	Layer 1 cache 失效重建，Layer 2 也失效。状态栏提示
用户上传/删除文档	同上
用户切换模型	所有 cache 失效重建，状态栏显示重建成本
摘要更新后 cache 变化	仅 Layer 2 重建，Layer 1 不受影响
Compaction API 触发	记录日志告警，状态栏显示"服务端已自动压缩"
Rolling Summary 超过 3000 tokens	对摘要本身再做一次 Haiku 压缩
Rolling Summary 不足 1024 tokens	不标记 cache_control，以普通文本发送
用户快速切换多对话触发多个压缩	全局请求队列，chat P0 > compress P1，限流时暂停压缩

第十一部分：开发阶段路线图

Phase 0: 环境与 API 验证（1-2 天）

1. 搭建 Python 3.12 venv，安装所有依赖
2. 编写最小脚本验证：(a) 流式调用 Sonnet 4.5；(b) cache_control 验证两次调用的 cache hit；(c) 代理连通性

Phase 1: 数据层 + 骨架 UI（1 周）

1. SQLite Schema 初始化（含压缩字段）
2. ProjectManager + ConversationManager CRUD
3. 三面板主窗口骨架
4. API Key 管理（keyring + Settings）

Phase 2: 核心聊天流程（1-2 周）

1. ClaudeClient 流式实现（QThread + asyncio）
2. ContextBuilder 四层构建
3. QWebEngineView Markdown 渲染（含 `js_safe()` 安全编码）
4. 完整流程打通：创建项目 → 新建对话 → 发送消息 → 流式显示 → 存入 DB
5. 模型选择器

Phase 3: Projects 功能（1-2 周）

1. 文档上传与文本提取
2. 系统提示词编辑器
3. Prompt Caching 集成（Layer 1 cache_control）
4. Token 追踪与成本显示
5. 多对话管理

Phase 4: Token 优化（1-2 周）

> **【专家建议采纳】** 不要在 Phase 2 就实现压缩逻辑。先完成基础聊天，再引入压缩。否则调试难度指数级上升。

1. 增量式滚动摘要系统（CompressionWorker + Haiku 压缩）
2. Layer 2 Cache Breakpoint 集成
3. Compaction API 兜底
4. Cache 命中率仪表盘
5. 成本预估修正（复用 ContextBuilder 逻辑）
6. 压缩策略预设 UI

Phase 5: 完善与打磨（1-2 周）

1. Extended Thinking 开关 + 预算滑块
2. 暗色/亮色主题 + 系统主题自动检测
3. 键盘快捷键

4. 图片附件支持
5. 对话搜索与导出（MD/JSON）
6. 错误恢复、重试 UI、代理配置

第十二部分：不做的事项（明确排除）

> **【专家建议评估后决定排除】**

排除项	理由
心跳保活机制	收益仅 5%，实现复杂，竞态风险高
基于关键词的模型自动推荐	规则引擎不可靠，且让用户感觉工具替自己做决定
发送前确认弹窗	高频使用场景下每次弹窗是灾难性 UX
本地 Embedding 模型	需 PyTorch 200MB+，与轻量级桌面应用定位不符
Cache 状态本地预测表	本地时钟预测服务端 TTL 天然不精确，可靠信号是 API 响应中的 cache_read_input_tokens
OCR（Phase 1-4）	复杂度高，Tesseract 需额外安装，作为 Phase 5+ 的 stretch goal
"记忆迷雾"可视化	UX 价值高但实现复杂，延后到 Phase 5+
Cache 预热（发送空请求）	增加不必要的 API 成本
完整文档传给 Haiku 做压缩	压缩是语言学概括，不需要领域知识，传文档会使成本飙升

第十三部分：验收标准

1. **成本验证**：50 轮对话场景下，累计输入 token 成本对比无压缩方案节省 50%+
2. **Cache 命中率**：正常对话节奏下，`cache_read_input_tokens` 占总缓存前缀 token 的 90%+
3. **压缩透明性**：压缩对用户不可见（无弹窗、无可感知延迟），仅在状态栏静默显示
4. **对话质量**：压缩后 Claude 仍能回忆早期对话中的关键决策和技术细节
5. **导出完整性**：Markdown 导出包含所有原始消息（非压缩版）
6. **容错性**：压缩失败时不影响主流程，自动回退到全文发送

第十四部分：依赖清单

```
# Core
anthropic>=0.40.0
PySide6>=6.6.0
PySide6-Addons>=6.6.0

# Document Processing
PyMuPDF>=1.24.0
pdfplumber>=0.11.0
python-docx>=1.1.0
openpyxl>=3.1.0
chardet>=5.0

# Token & Cost
tiktoken>=0.7.0

# Security
keyring>=25.0

# UI Rendering
markdown>=3.5
pymdown-extensions>=10.0
```

```
Pygments>=2.17

# Image
Pillow>=10.0

# Utilities
darkdetect>=0.8.0
httpx[socks]>=0.27.0

# Development
ruff>=0.4.0
mypy>=1.10
```

附录 A：专家审核意见处理清单

专家建议	处置	本文档位置
异步预压缩（CompressionWorker）	✅ 采纳	§2.2，§4.4.4
代码块保护压缩提示词	✅ 采纳	§4.4.5
成本预估逻辑冲突修正	✅ 采纳（关键）	§4.7
ContextBuilder 四层重写	✅ 采纳（关键）	§4.3，§4.6
DB Schema 合并	✅ 采纳	§3.1
1024 token 缓存门槛	✅ 采纳	§4.3 备注
缓存链式失效文档化	✅ 采纳	§4.3 备注，§5.2
压缩失败降级处理	✅ 采纳	§4.8
限流优先级（chat > compress）	✅ 采纳	§7.1

专家建议	处置	本文档位置
压缩策略预设	✅ 采纳（简化为 2 档）	§ 4.10
重置摘要按钮	✅ 采纳	§ 4.4.6, § 8.4
状态栏压缩反馈	✅ 采纳	§ 8.3
Haiku 上下文盲区	✅ 决策：不传文档	§ 4.4.5 备注
记忆迷雾可视化	⏸ 延后至 Phase 5+	§ 12
Cache 预热空请求	❌ 排除	§ 12
心跳保活	❌ 排除	§ 4.1, § 12

文档结束。本文档涵盖了 ClaudeStation 从架构设计到实现细节的完整规格。开发时以本文档为唯一基准，三份原始文档仅作参考。