数据分析智能体多端开发项目 - 详细设计文档

一、文档概述

1.1 文档目的

本文档用于明确“数据分析智能体多端开发项目”的核心架构、技术选型、模块设计、接口规范、数据库设计及安全策略，为后端开发、前端适配、前后端联调及后续迭代提供标准化依据，确保项目团队成员对开发方案认知一致。

1.2 适用范围

覆盖项目全链路开发，包括后端服务（模型服务、API 服务、数据库服务）、前端 Flutter 多端适配、数据交互流程、安全防护及部署运维等环节，适用于项目开发人员、测试人员及运维人员。

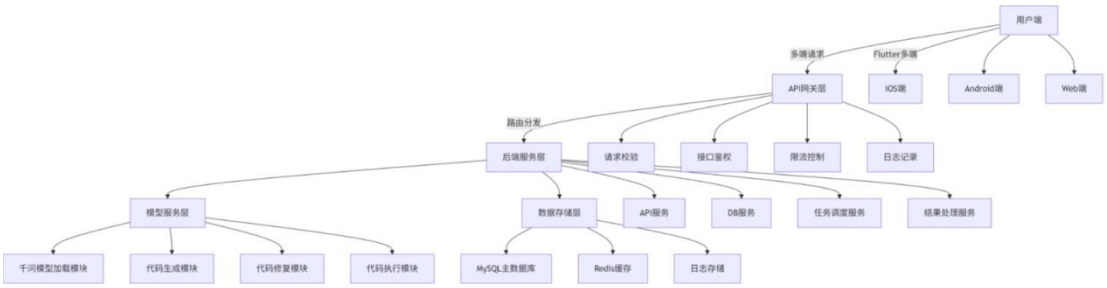
1.3 术语定义

术语	定义
LLM	大语言模型（Large Language Model），本文档特指千问 Qwen-1.8B-Chat
Tokenizer	分词器，用于将自然语言文本转换为模型可识别的 token 序列
API 服务	提供前后端数据交互的接口服务，基于 FastAPI 实现
DB 服务	数据库操作服务，封装数据增删改查逻辑，基于 SQLAlchemy ORM 实现
沙箱环境	隔离的代码执行环境，用于防止恶意代码执行影响系统安全
多端适配	基于 Flutter 实现 iOS、Android、Web 多端统一体验

二、项目整体架构设计

2.1 架构概述

项目采用“前后端分离+微服务拆分”架构，核心分为前端层、API 网关层、后端服务层、数据存储层及模型服务层，各层职责清晰、松耦合，便于后续扩展和维护。整体架构图如下：



2.2 核心架构原则

- 松耦合：各模块独立部署、独立迭代，通过标准化接口通信，降低模块间依赖
- 高可用：核心服务支持冗余部署，数据库采用主从备份，避免单点故障
- 安全性：全链路接口鉴权，代码执行隔离，敏感数据加密存储，防止恶意攻击
- 可扩展性：架构设计预留扩展接口，支持后续新增模型（如 Qwen-7B）、新增功能（如数据可视化）
- 易用性：前端交互简洁直观，后端接口设计符合 RESTful 规范，降低开发和使用成本

2.3 技术栈选型

层级	技术选型	选型理由
前端层	Flutter + GetX 状态管理	一套代码适配多端（iOS/Android/Web），开发效率高，性能接近原生
API 网关层	FastAPI 中间件	轻量高效，支持异步请求，原生支持 OpenAPI 文档，便于接口管理
后端服务层	Python 3.9 + FastAPI	Python 生态丰富，适配 LLM 模型开发；FastAPI 性能优异，支持异步处理高并发请求
模型服务层	Transformers + Torch + Safetensors	Transformers 支持千问模型快速加载，Torch 适配 CPU/GPU 部署，Safetensors 提升模型加载安全性
数据存储层	MySQL 8.0 + Redis 6.0	MySQL 支持复杂查询，适配业务数据存储；Redis 用于缓存热点数据（如用户 token），提升响应速度
开发/运维工具	Git（版本控制）+ Docker（容器化）+ Nginx（反向代理）	Docker 实现环境一致性，Nginx 实现负载均衡，降低部署和运维成本

三、核心模块详细设计

3.1 模型服务层设计

模型服务层是项目核心，负责千问模型加载、中文需求解析、Python 代码生成、代码语法修复及代码安全执行，核心分为 4 个模块：

3.1.1 千问模型加载模块

核心职责

实现千问 Qwen-1.8B-Chat 模型的本地离线加载，解决模型加载过程中的各类报错（如 pad_token 缺失、unknown ids 等），确保模型稳定可用。

核心设计

- 模型路径配置：通过配置文件指定本地模型路径，支持环境变量动态注入，适配不同部署环境
- 特殊标记处理：硬编码千问官方默认 eos_token_id=151643，复用为 pad_token_id，禁用 padding，彻底规避特殊标记相关报错
- 加载优化：采用 low_cpu_mem_usage=True 参数，降低 CPU 内存占用；模型加载后缓存至内存，避免重复加载
- 异常处理：捕获模型加载过程中的文件缺失、依赖冲突等异常，输出详细错误日志，便于问题排查

核心代码框架

```
python
def load_local_qwen_model():
    global _tokenizer, _model
    if _tokenizer is None or _model is None:
        try:
            # 1. 配置参数加载
            model_path = config.MODEL_LOCAL_PATH
            qwen_eos_token_id = config.QWEN_EOS_TOKEN_ID

            # 2. 核心文件校验
            required_files = ["config.json", "modeling_qwen.py",
"model.safetensors.index.json", "tokenizer_config.json"]
            missing_files = [f for f in required_files if not
os.path.exists(os.path.join(model_path, f))]
            if missing_files:
                raise FileNotFoundError(f"缺失核心文件：
{missing_files}")

            # 3. Tokenizer 加载与配置
            _tokenizer = AutoTokenizer.from_pretrained(
                model_path,
                trust_remote_code=True,
                padding_side="left",
                truncation_side="left",
                local_files_only=True,
                use_fast=False,
                add_bos_token=False,
                add_eos_token=False
            )
            _tokenizer.eos_token_id = qwen_eos_token_id
            _tokenizer.pad_token_id = qwen_eos_token_id

            # 4. 模型加载
            _model = AutoModelForCausalLM.from_pretrained(
```

```
                model_path,
                trust_remote_code=True,
                device_map="cpu",
                torch_dtype=torch.float32,
                low_cpu_mem_usage=True,
                load_in_8bit=False,
                local_files_only=True,
                use_safetensors=True,
                ignore_mismatched_sizes=True
            ).eval()

            logger.info("千问模型加载完成")
        except Exception as e:
            logger.error(f"模型加载失败：{str(e)}")
            raise
    return _tokenizer, _model
```

3.1.2 代码生成模块

核心职责

接收前端传入的中文数据分析需求，通过 Prompt 工程引导千问模型生成符合要求的 Python 代码，确保代码简洁、可运行、无多余内容。

核心设计

- Prompt 模板设计：采用极简模板，明确要求模型仅输出可运行 Python 代码，无解释、注释、markdown 标记，强调缩进规范
- 生成参数优化：设置 temperature=0.2（降低随机性）、top_p=0.9（提升生成稳定性），max_new_tokens=512（控制生成长度）
- 输入预处理：对用户需求进行清洗，过滤特殊字符，避免干扰模型生成
- 输出过滤：移除生成文本中的原始 Prompt，过滤空行、注释、markdown 标记等无效内容

核心 Prompt 模板

```
python
prompt_template = """
生成 Python 代码实现以下数据分析需求：{requirement}
要求：
1. 严格遵守 Python 缩进规范（4 个空格缩进），确保 if/for/def 等块语句后有正确缩进的代码块；
2. 仅输出可运行的 Python 代码，无任何解释、注释、markdown 标记（如```）；
3. 优先使用内置库（math/statistics/random），如需第三方库需标注，但优先保证代码简洁；
4. 代码需包含完整的输入、计算、打印输出步骤，变量名使用英文，适配 Python 3.8+。
```

```
"""
```

3.1.3 代码修复模块

核心职责

针对模型生成代码中可能存在的缩进错误（如块语句后无缩进、嵌套缩进错误），通过语法分析实现自动修复，提升代码可运行性。

核心设计

- 基于抽象语法树（AST）的修复逻辑：利用 Python 内置 ast 模块解析生成代码，识别缩进错误节点，自动补全正确缩进
- 块语句识别：精准识别 if、for、while、def、class 等块语句，处理单层及嵌套块的缩进层级
- 修复容错：若 AST 解析失败（代码语法错误严重），返回空字符串，触发失败流程

核心代码框架

```python
def fix_code_indentation(code):
    """基于 AST 的代码缩进修复"""
    if not code:
        return ""
    try:
        # 1. 解析代码为 AST
        tree = ast.parse(code)
        # 2. 遍历 AST，识别块语句节点
        indent_level = 0
        indent_step = 4
        fixed_lines = []

        for node in ast.walk(tree):
            if isinstance(node, (ast.If, ast.For, ast.While,
ast.FunctionDef, ast.ClassDef)):
                # 处理块语句开始，增加缩进
                fixed_lines.append(" " * indent_level *
indent_step + ast.unparse(node.body[0].lineno-1))
                indent_level += 1
            elif isinstance(node, ast.Pass):
                # 处理 pass 语句，保持当前缩进
                fixed_lines.append(" " * indent_level *
indent_step + "pass")
            # 其他节点处理...

        return "\n".join(fixed_lines)
    except SyntaxError:
        logger.error("代码语法错误，无法修复")
        return ""
```

3.1.4 代码执行模块

核心职责

在隔离的沙箱环境中执行修复后的 Python 代码，捕获代码执行结果和异常信息，确保代码执行不会影响系统安全。

核心设计

- 沙箱隔离：使用 subprocess 模块创建独立进程执行代码，限制进程权限，禁止访问系统敏感资源（如文件系统、网络）
- 超时控制：设置代码执行超时时间（默认 10 秒），避免恶意代码（如死循环）占用系统资源
- 结果捕获：重定向标准输出和标准错误，捕获代码执行结果和异常信息，便于后续存储和展示
- 安全校验：执行前对代码进行安全检查，过滤危险操作（如 os.system、eval、exec 等）

3.2 后端服务层设计

3.2.1 API 服务模块

核心职责

提供前后端数据交互的标准化 API 接口，实现请求校验、接口鉴权、业务逻辑处理及响应返回，遵循 RESTful 规范。

核心接口设计

接口路径	请求方法	请求参数	响应参数	接口描述
/api/user/login	POST	{"username": string, "password": string}	{"code": 200, "msg": "success", "data": {"token": string, "user_id": int}}	用户登录，获取访问 token
/api/code/generate	POST	{"requirement": string, "user_id": int}	{"code": 200, "msg": "success", "data": {"task_id": int, "generated_code": string}} / {"code": 400, "msg": "fail", "data": ""}	提交数据分析需求，生成 Python 代码
/api/code/execute	POST	{"task_id": int, "code": string}	{"code": 200, "msg": "success", "data": {"output": string, "status": "success"}} / {"code": 400, "msg": "fail", "data": {"error_msg": string}}	执行生成的 Python 代码，返回执行结果
/api/task/history	GET	{"user_id": int, "page": int,	{"code": 200, "msg": "success", "data": {"total": int, "tasks":	查询用户历史任务

		"page_size": int}	[{"task_id": int, "requirement": string, "create_time": string, "status": string}]}}	记录
/api/task/detail	GET	{"task_id": int}	{"code": 200, "msg": "success", "data": {"task_id": int, "requirement": string, "generated_code": string, "output": string, "create_time": string}}	查询单个任务的详细信息

接口鉴权设计

采用 JWT（JSON Web Token）实现接口鉴权，流程如下：

1. 用户登录时，后端验证用户名密码正确后，生成 JWT Token（包含 user_id、过期时间等信息）返回给前端
2. 前端后续请求需在 HTTP 请求头中携带 Token：Authorization: Bearer {Token}
3. 后端 API 网关层验证 Token 的有效性和过期时间，验证通过则路由到对应服务，验证失败返回 401 未授权
4. Token 过期时间设置为 2 小时，前端可通过刷新 Token 接口获取新的 Token

3.2.2 DB 服务模块

核心职责

封装数据库增删改查（CRUD）操作，实现数据持久化存储，为其他服务提供数据访问支持，确保数据操作的安全性和一致性。

核心设计

1.ORM 映射：基于 SQLAlchemy 实现数据库表与 Python 类的映射，简化数据库操作
2.连接池配置：使用数据库连接池管理连接，设置最大连接数、空闲连接数等参数，提升数据库访问效率
3.事务管理：对涉及多表操作的业务逻辑（如创建任务+记录日志）开启事务，确保数据一致性
4.异常处理：捕获数据库操作异常（如连接失败、SQL 语法错误），输出详细日志，便于问题排查

3.2.3 任务调度服务模块

核心职责

管理用户提交的数据分析任务，实现任务的创建、状态更新、超时处理等功能，确保任务有序执行。

核心设计

- 任务状态设计：定义任务状态流转：待处理→生成中→执行中→成功/失败
- 超时处理：对长时间未完成的任务（如生成代码超时、执行代码超时）进行自动终止，更新任务状态为失败

- 任务优先级：支持设置任务优先级（默认普通优先级），高优先级任务优先执行

3.3 数据存储层设计

3.3.1 数据库选型与架构

采用“MySQL 主从+Redis 缓存”的存储架构：

- MySQL 主库：存储核心业务数据（用户信息、任务记录、执行日志等），支持复杂查询和事务
- MySQL 从库：同步主库数据，用于读取操作（如历史任务查询），分担主库压力
- Redis 缓存：缓存热点数据（如用户 Token、近期任务记录），提升查询响应速度

3.3.2 核心表结构设计（MySQL）

1. user 表（用户信息表）

字段名	字段类型	是否主键	默认值	备注
id	int(11)	是	auto_increment	用户唯一 ID
username	varchar(50)	否	无	用户名，唯一
password	varchar(100)	否	无	密码，MD5 加密存储
email	varchar(100)	否	无	用户邮箱，可选
create_time	datetime	否	current_timestamp	用户创建时间
update_time	datetime	否	current_timestamp on update current_timestamp	用户信息更新时间
status	tinyint(1)	否	1	用户状态：1-正常，0-禁用

2. task 表（任务记录表）

字段名	字段类型	是否主键	默认值	备注
id	int(11)	是	auto_increment	任务唯一 ID
user_id	int(11)	否	无	外键，关联 user 表 id
requirement	text	否	无	用户提交的数据分析需求

generated_code	text	否	无	模型生成的 Python 代码
status	varchar(20)	否	pending	任务状态：pending-待处理，generating-生成中，executing-执行中，success-成功，fail-失败
priority	varchar(10)	否	normal	任务优先级：high-高，normal-普通，low-低
create_time	datetime	否	current_timestamp	任务创建时间
update_time	datetime	否	current_timestamp on update current_timestamp	任务状态更新时间

3. execution_log 表（代码执行日志表）

字段名	字段类型	是否主键	默认值	备注
id	int(11)	是	auto_increment	日志唯一 ID
task_id	int(11)	否	无	外键，关联 task 表 id
output	text	否	无	代码执行输出结果
error_msg	text	否	无	代码执行错误信息（失败时非空）
execute_time	datetime	否	current_timestamp	代码执行时间
execute_duration	float	否	无	代码执行耗时（秒）

3.3.3 Redis 缓存设计

缓存 Key	缓存 Value	过期时间	备注
user:token:{user_id}	JWT Token 字符串	2 小时	缓存用户登录 Token，用于快速验证
task:latest:{user_id}	最近 5 条任务记录 JSON	30 分钟	缓存用户近期任务，提升查询速度
model:load:status	1-已加载，0-未加载	永久	缓存模型加载状态，避免重复检查

四、安全设计

● 4.1 接口安全

- 接口鉴权：所有核心接口采用 JWT 鉴权，未携带有效 Token 的请求直接拒绝
- 请求校验：对接口入参进行合法性校验（如数据类型、长度、格式），过滤非法请求

- 限流控制：基于用户 ID 和 IP 地址实现接口限流，避免恶意请求攻击（如每秒最多 5 次请求）
- HTTPS 加密：生产环境采用 HTTPS 协议传输数据，防止数据在传输过程中被窃取或篡改

4.2 代码执行安全

- 沙箱隔离：使用独立进程执行代码，限制进程资源（CPU、内存），禁止访问系统敏感资源
- 代码过滤：执行前过滤危险操作，如文件读写（os.open、open）、网络请求（requests）、系统命令（os.system、subprocess）等
- 超时控制：设置代码执行超时时间（默认 10 秒），超时自动终止进程，释放资源

4.3 数据安全

- 敏感数据加密：用户密码采用 MD5 加密存储，禁止明文存储；核心业务数据（如用户 Token）在 Redis 中加密存储
- 数据库权限控制：数据库用户仅分配必要的权限（如应用用户仅拥有增删改查权限，无创建/删除表权限）
- 数据备份：MySQL 主库定期备份数据（每日凌晨全量备份，每小时增量备份），防止数据丢失

五、部署设计

5.1 部署架构

采用 Docker 容器化部署，配合 Nginx 实现负载均衡，部署架构如下：

[流程图暂不支持下载]

5.2 部署环境要求

环境类型	CPU	内存	硬盘	操作系统
开发环境	4 核	8G	100G	Windows 10/11 或 Ubuntu 20.04
测试环境	8 核	16G	200G	Ubuntu 20.04
生产环境	16 核	32G	500G SSD	Ubuntu 20.04 LTS

5.3 部署步骤（简化）

1. 安装 Docker 和 Docker Compose
2. 拉取项目代码，配置环境变量（如模型路径、数据库信息）
3. 执行 docker-compose up -d 启动所有服务容器
4. 初始化数据库（执行建表 SQL 脚本）

5. 配置 Nginx 反向代理和负载均衡规则
6. 验证服务可用性（访问前端页面、调用 API 接口）

六、项目迭代规划

6.1 第一阶段：基础功能实现（1-2 周）

- 完成千问模型本地部署和代码生成核心功能
- 实现 API 服务核心接口（代码生成、任务查询）
- 完成数据库建表和基础 DB 服务封装
- 实现前端基础页面（需求输入、结果展示）

6.2 第二阶段：功能完善（2-3 周）

- 优化代码修复逻辑，解决复杂嵌套代码缩进问题
- 实现代码执行沙箱隔离和安全校验
- 完成前后端联调，实现全流程功能闭环
- 实现用户登录、权限控制功能

6.3 第三阶段：性能优化与扩展（3-4 周）

- 优化模型加载和代码生成速度（如模型量化、缓存优化）
- 实现数据库主从分离和 Redis 缓存，提升系统性能
- 开发数据可视化功能（对接前端图表组件）
- 支持多模型切换（如 Qwen-7B、ChatGLM）

6.4 第四阶段：测试与部署（1-2 周）

- 完成系统全面测试（功能测试、性能测试、安全测试）
- 修复测试过程中发现的问题
- 完生产环境部署和运维文档编写

七、风险与应对措施

风险类型	风险描述	应对措施
技术风险	千问模型 CPU 部署生成质量不足，复杂代码缩进错误无法彻底解决	1. 尝试模型量化或升级到 Qwen-7B 模型；2. 引入专业 Python 语法解析库提升修复精度；3. 限制复杂需求的支持范围，优先保证简单需求的稳定性
性能风险	高并发场景下，模型生成和代码执行速度慢，系统响应延迟	1. 实现模型和代码生成结果缓存；2. 采用分布式部署，增加服务节点；3. 优化代码执行效率，使用异步处理
安全	恶意用户提交危险代码，	1. 加强代码安全校验，过滤所有危险操作；2. 严格限

风险	攻击系统安全	制沙箱环境权限；3. 实现接口限流和恶意请求识别
进度风险	核心功能开发周期超出预期，影响项目上线	1. 明确优先级，优先实现核心功能；2. 合理分配开发资源，并行开发不同模块；3. 定期跟进进度，及时调整开发计划

八、附录

8.1 依赖库清单

```txt
# 后端核心依赖
torch==2.0.1
transformers==4.35.0
safetensors==0.4.0
fastapi==0.103.1
uvicorn==0.23.2
sqlalchemy==2.0.20
redis==5.0.1
python-jose==3.3.0
passlib==1.7.4

# 前端核心依赖
flutter==3.13.0
getx==4.6.5
dio==5.3.2
fl_chart==0.55.2
```

8.2 相关文档参考

- 千问模型官方文档：https://help.aliyun.com/document_detail/2517353.html
- FastAPI 官方文档：https://fastapi.tiangolo.com/
- SQLAlchemy 官方文档：https://docs.sqlalchemy.org/
- Flutter 官方文档：https://docs.flutter.dev/

附件

姓名	徐航毅
任务	写代码

AI参考	# feature/server/app/services/llm_service.py
	import sys
	import os
	sys.path.append(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(__file__)))))
	
	from transformers import AutoTokenizer, AutoModelForCausalLM
	import torch
	import ast
	
	# 模型配置（根据实际路径调整）
	MODEL_PATH = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(__file__)))), "llm/models/Qwen-1_8B-Chat")
	QWEN_EOS_TOKEN_ID = 151643
	
	# 全局模型/分词器缓存
	_tokenizer = None
	_model = None
	
	def load_local_qwen_model():
	  """加载本地千问模型（单例模式）"""
	  global _tokenizer, _model
	  if _tokenizer is None or _model is None:
	  try:
	  # 加载分词器
	  _tokenizer = AutoTokenizer.from_pretrained(
	  MODEL_PATH,
	  trust_remote_code=True,
	  padding_side="left",
	  truncation_side="left",
	  local_files_only=True,
	  use_fast=False,
	  add_bos_token=False,
	  add_eos_token=False
	  )
	  _tokenizer.eos_token_id = QWEN_EOS_TOKEN_ID
	  _tokenizer.pad_token_id = QWEN_EOS_TOKEN_ID
	
	  # 加载模型
	  _model = AutoModelForCausalLM.from_pretrained(
	  MODEL_PATH,
	  trust_remote_code=True,
	  device_map="cpu",
	  torch_dtype=torch.float32,
	  low_cpu_mem_usage=True,

```
            load_in_8bit=False,
            local_files_only=True,
            use_safetensors=True,
            ignore_mismatched_sizes=True
        ).eval()
        print("千问模型加载成功")
    except Exception as e:
        raise Exception(f"模型加载失败：{str(e)}")
  return _tokenizer, _model

def fix_code_indentation(code: str) -> str:
  """基于 AST 修复代码缩进"""
  if not code:
    return ""
  try:
    # 解析为 AST
    tree = ast.parse(code)
    indent_level = 0
    indent_step = 4
    fixed_lines = []

    # 遍历 AST 节点修复缩进（简化版，适配基础块语句）
    for node in ast.walk(tree):
      if isinstance(node, (ast.If, ast.For, ast.While, ast.FunctionDef, ast.ClassDef)):
        # 块语句开始，增加缩进
        fixed_lines.append("    " * indent_level * indent_step + ast.unparse(node).split("\n")[0])
        indent_level += 1
      elif isinstance(node, ast.Pass):
        fixed_lines.append(" " * indent_level * indent_step + "pass")
      elif isinstance(node, ast.Expr):
        fixed_lines.append(" " * indent_level * indent_step + ast.unparse(node))

    return "\n".join(fixed_lines)
  except SyntaxError:
    # 解析失败返回原代码
    return code

def generate_code_from_requirement(requirement: str) -> str:
  """根据中文需求生成 Python 代码"""
  try:
    # 1. 加载模型
    tokenizer, model = load_local_qwen_model()

    # 2. 构建 Prompt
    prompt = f"""
生成 Python 代码实现以下数据分析需求：{requirement}
要求：
1. 严格遵守 Python 缩进规范（4 个空格缩进）；
2. 仅输出可运行的 Python 代码，无任何解释、注释、markdown 标记；
3. 优先使用内置库（math/statistics/random）；
4. 代码包含完整的输入、计算、打印输出步骤。
    """.strip()
```

```
        # 3. 编码输入
        inputs = tokenizer(
            prompt,
            return_tensors="pt",
            truncation=True,
            max_length=1024,
            padding=False
        )

        # 4. 生成代码
        with torch.no_grad():
            outputs = model.generate(
                **inputs,
                max_new_tokens=512,
                temperature=0.2,
                top_p=0.9,
                do_sample=True,
                eos_token_id=QWEN_EOS_TOKEN_ID,
                pad_token_id=QWEN_EOS_TOKEN_ID
            )

        # 5. 解码输出
        generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)

        # 6. 过滤 Prompt，提取代码
        code = generated_text.replace(prompt, "").strip()
        # 移除 markdown 代码块标记（若有）
        if code.startswith("```python"):
            code = code.replace("```python", "").replace("```", "").strip()

        # 7. 修复缩进
        fixed_code = fix_code_indentation(code)
        return fixed_code
    except Exception as e:
        raise Exception(f"代码生成失败：{str(e)}")

# 测试代码
if __name__ == "__main__":
    # 测试生成代码
    requirement = "计算 1 到 100 的累加和"
    code = generate_code_from_requirement(requirement)
    print("生成的代码：")
    print(code)
```

姓名	张赛
任务	AI 收集资料
AI 参考	**数据分析智能体多端开发详细设计报告（基于千问模型）** **一、项目概述** **（一）项目背景** 高校学生在课程作业、学科竞赛及学生工作中高频面临数据分析需求，但现有工具存

	在明显痛点：专业工具（Python、SPSS、R）学习门槛高，轻量化工具功能有限，且多数工具局限于单一终端，跨设备数据同步不便。结合生成式 AI 技术发展趋势，Gartner 2024 年数据显示大模型在数据分析领域的落地渗透率已达 42%，可使分析链路缩短 60%。本项目基于阿里云通义千问大模型，开发支持网页端、移动端 APP、桌面软件的多端数据分析智能体，通过自然语言交互降低数据分析门槛，同时满足学生多样化使用场景需求。 **（二）项目目标** 1. 基于千问模型实现“自然语言需求→智能分析→多形式输出”全流程，支持 Python/SPSS/R 代码生成与直接分析结论双模式输出。 2. 整合文件上传、联网检索、基础爬虫三大数据输入方式，适配学生常见数据来源场景。 3. 实现多端数据实时同步与一致化管理，支持离线缓存与历史记录回溯。 4. 8 周内完成 MVP 版本开发，确保在学生常用硬件环境下运行流畅，核心功能响应时间≤15 秒。 5. 基于千问模型的轻量化部署与扩展能力，预留功能迭代接口，支持后续新增分析工具与场景适配。 **（三）项目范围** **1. 包含功能范围** • 支持 CSV/Excel 文件上传解析（单个文件≤10MB），中文数据无乱码处理。 • 千问模型驱动的自然语言需求解析，拒绝非数据分析类无效请求。 • 折线图、柱状图、饼图、散点图生成与 PNG 格式导出（分辨率≥1080×720px）。 • 联网检索公开数据（如国家统计局宏观数据）、静态网页表格爬虫（遵守 robots 协议）。 • 多端数据增量同步，冲突时提供用户自主选择策略。 • 分析记录管理（查询、删除、再次分析）与 Word 格式报告导出。 **2. 暂不支持功能** • 复杂机器学习模型训练（如回归、聚类算法）与动态网页爬虫。 • 多用户协作共享、在线支付及跨区域大规模数据同步。 • 自定义图表样式、复杂数据清洗与高并发访问支持。 • 付费第三方 API 调用与短信验证等增值服务。

姓名	蒋雨霏
任务	撰写详细设计文档
AI 参考	略

姓名	刘畅
任务	下载千问 G3-M 模型
AI 参考	略

姓名	玉曦
任务	创建 MySQL 账号密码
AI 参考	略