

详细设计文档

1. 1. 文档概述（Overview）

1.1 文档目的（Purpose）

本规范用于定义本项目中 **从自然语言问题（NL）到最终回答（Answer）** 的完整工程流水线，包括：

NL → subqueries → PLAN → SQL → Result → Answer

它给出各阶段的输入/输出、职责边界和调用顺序，为以下工作提供统一的“蓝本”：

- 代码实现与重构（模块划分、接口定义、异常处理）
- 代码评审与设计评审（是否符合既定流程与约束）
- 后续测试、评测、监控与排查（以阶段为单位定位问题）

本规范只约束 **服务端 NL2SQL 流程**，即：

- FastAPI 服务、流水线各阶段组件、SQL 生成与执行、与 LLM 的交互方式
- 不涉及前端 UI 展示、业务 BI 报表配置、图表样式等内容

前端或上层 ChatFlow 可以把本服务视为一个“黑盒能力”：输入自然语言问题和用户上下文，获取结构化结果与自然语言回答。

1.2 适用范围与读者（Scope & Audience）

适用项目

- 当前 NL2SQL 服务端工程（基于 FastAPI 的后端服务）
- 在该工程基础上的后续版本与扩展（增加业务域、增加数据源、更换 LLM 等）

预期读者

- **后端开发 / 数据工程**
 - 根据本规范拆分模块，实现/调整各阶段逻辑
 - 保证接口签名、输入输出结构和异常行为与规范一致
- **测试 / 质量工程**
 - 基于阶段定义设计用例、构造评测集和集成测试
 - 按阶段定位问题属于 NL 解析、Plan 生成、SQL 生成还是执行层
- **运维 / SRE**
 - 结合本规范理解各阶段的日志与监控指标

- 基于阶段化信息建立报警规则和排查手册
- **产品经理 / 架构师**
 - 从整体视角理解系统能力边界、交互方式和限制
 - 在规划新需求时判断应该落在哪个阶段、是否需要扩展语义层或 PLAN 模型

1.3 MVP 阶段负向约束 (Negative Constraints)

为了确保 V1 版本快速稳定上线，以下场景明确列为**不支持**，后端开发需在代码中设置“熔断检查”，遇到此类情况直接抛出 `UNSUPPORTED_FEATURE` 异常：

1. 跨事实表查询 (No Drill-across):

- **定义：**一次查询中包含的指标 (Metrics) 来自不同的实体 (Entity)，且这些实体之间没有直接的维度表关联。
- **行为：**Stage 3 校验时，若发现 `metrics` 对应的 `entity_id` 不唯一，直接报错。

2. 复杂嵌套逻辑 (No Nested Queries):

- **定义：**如“查询销售额大于平均值的城市”。
- **行为：**Stage 1 应将其拆解为两个步骤，Stage 2 仅处理单层平铺的查询。PLAN 结构不支持递归。

3. 自定义时间偏移 (No Custom Offset):

- **定义：**如“向前推 3.5 天”、“对比上周二”。
- **行为：**仅支持 `YOY` (同比) 和 `MOM` (环比)。其他时间偏移需求将被忽略或报错。

4. 数据写入与修改 (Read-Only):

- **定义：**任何 `INSERT`，`UPDATE`，`DELETE` 意图。
- **行为：**Stage 4 生成 SQL 时严禁包含 DML 语句，Stage 5 数据库连接强制开启只读模式。

5. 有限的数据库方言支持 (Limited SQL Dialects):

- **定义：**MVP 版本仅官方支持 **MySQL (8.0+)** 和 **PostgreSQL (13+)**。
- **行为：**`PipelineConfig` 中的 `db_type` 若配置为其他值（如 Oracle, Sqlite），系统将在启动时直接抛出 `CONFIGURATION_ERROR`。Stage 4/5 的适配器仅实现上述两种方言的逻辑分支。

6. 物理连接屏蔽 (No Physical Joins / Semantic View Only):

- **定义：**MVP 阶段，NL2SQL 生成器不负责推导复杂的实体间 Join 路径（如 A join B join C）。
- **行为：**系统假设所有查询意图都能在 **语义视图 (Semantic Views)** 层面解决。Stage 4 生成 SQL 时，FROM 子句仅指向 YAML 中配置的 `semantic_view`（宽表），严禁动态生成 `JOIN` 子句。若需求超出宽表范围，视为不支持。

7. 严格的数据库类型约束 (Strict Database Schema Types):

- **定义:** 底层数据库中涉及时间计算的字段 (如 `order_date`) , 必须定义为标准的 `DATE` 或 `DATETIME/TIMESTAMP` 类型。
- **行为: 不支持** 使用字符串 (`VARCHAR`) 存储日期。Stage 4 生成的 SQL 依赖 `DATE_TRUNC` / `DATE_FORMAT` 等函数, 这些函数对字符串类型字段无效。

1.4 相关文档引用 (Related Documents)

本规范关注的是 “**时序流程与阶段职责**”, 不重复定义结构和规则。下面是与之强相关的其他文档:

- **《PLAN 模型结构定义》**
 - 定义 Plan Schema 的字段结构与枚举约束 (注: 具体的兜底业务逻辑由本文档的第3章阶段三中承载) 。
 - 定义 NL→PLAN 的语义解析规则
 - 在本规范中, 所有涉及 PLAN 字段的描述均以该文档为准
- **《语义层配置与 YAML Schema 规范》**
 - 定义 `semantic_core.yaml`、`semantic_metrics.yaml`、`semantic_security.yaml`、`term_synonyms.yaml`、`enum_dicts.yaml` 等配置结构
 - 描述业务域、实体、指标、维度、枚举、权限在配置层的落地方式
 - 本规范中涉及 “如何从配置中查 domain/entity/metric/dimension/security” 的部分, 仅描述调用方式, 不重述配置结构
- **《安全与权限设计规范 (Row/Column Security for NL2SQL) 》**
 - 定义行级/列级权限模型、角色与范围的映射规则
 - 描述权限在 PLAN、SQL、Result 各阶段的注入与校验方式
 - 本规范仅说明 “在流水线的哪些阶段挂载安全逻辑”, 具体策略以该文档为准
- **《NL2SQL 评测与测试规范》**
 - 定义评测集、评估指标 (可执行率、正确率、延迟等) 及测试流程
 - 本规范中提到的 “阶段成功率、错误分类” 等指标, 在详细定义上依赖该文档

关系说明:

- 本规范是 “**流程视角**”: 描述一次请求如何跨阶段流转。
- 其它三个规范分别从 **结构 (PLAN)**、**配置 (语义层 & 权限)**、**质量 (评测)** 的角度进行约束。
- 代码实现应同时满足:

- 流程行为符合本规范；
- 中间数据结构符合 PLAN 规范；
- 所有语义相关内容从 YAML 语义层读取；
- 权限与安全行为符合安全规范；
- 质量与测试行为符合评测规范。

2. 总体架构与设计原则 (Architecture & Principles)

2.1 端到端架构概览 (End-to-End Architecture)

本项目的端到端架构可以抽象描述为：

Client → NL2SQL API → Pipeline Stages → Semantic Views (DB) → Result → Answer

- **Client (客户端)**

- 可以是 Postman、命令行脚本、上层 ChatFlow、未来的 Web 前端组件等。
- 通过 HTTP 调用 NL2SQL 服务的接口（例如 `/nl2sql/execute`）。

- **NL2SQL API (接口层)**

- 基于 FastAPI 实现，负责：
 - 接收 HTTP 请求，解析入参、用户上下文
 - 分发到对应的流水线入口（例如 `execute / plan / sql`）
 - 将流水线输出封装为标准 HTTP 响应

- **Pipeline Stages (流水线阶段)**

本规范的核心部分，包含以下6个阶段：

- a. Request & Context 处理
- b. subqueries 拆解
- c. NL → PLAN 解析
- d. PLAN 校验与标准化
- e. PLAN → SQL 生成与执行
- f. Result → Answer 生成

- **Semantic Views (DB) (数据库与语义视图)**

- 由数据仓库/主题库提供底层数据（事实表、维表、视图）；
- 对 NL2SQL 服务而言，主要访问语义视图（如 `v_sales_order_item`），减少对底层物理表的耦合。

- **Result（结构化结果）**

- 来自数据库执行 SQL 的结果集，经过标准化处理，保留为结构化数据（行、列、类型等）。

- **Answer（自然语言回答）**

- 基于 Result 和原始问题，由 LLM 或模板生成的自然语言总结，用于直接反馈给用户。

在实现层面，上述组件部署在同一后端服务中：API 层负责对外提供接口，流水线与语义层、权限模块在内部协同工作，通过数据库连接池访问数据源。

2.2 阶段划分（Stages Definition）

为了避免“巨型函数”和职责混乱，端到端流程在工程上被划分为 6 个主阶段：

1. Request & Context（请求与上下文处理）

- 主要职责：
 - 接收原始自然语言问题与请求参数
 - 构建统一的请求上下文对象（包含 user_id、role_id、locale、timezone、trace_id 等）
- 不做任何语义解析与 SQL 逻辑，仅完成“输入规范化”。

2. subqueries 拆解（NL → subqueries）

- 主要职责：
 - 判断当前问题是否需要拆解为多个子问题
 - 如需要，生成一组 subqueries，每个 subquery 是可以独立生成 PLAN 的原子问题
- 不负责具体指标/维度识别，不直接生成 PLAN。

3. NL → PLAN 解析（subquery → PLAN）

- 主要职责：
 - 对每个 subquery 进行语义解析，生成对应的 PLAN 对象
 - 使用语义层配置（域、实体、指标、维度、枚举等）与规则，完成自然语言到结构化查询计划的映射
- 不直接访问数据库，不负责 SQL 字符串生成。

4. PLAN 校验 & 标准化（Plan Validation & Normalization）

- 主要职责：
 - 对 PLAN 进行结构与语义校验（字段完整性、合法性、组合合理性）
 - 根据兜底规则补全缺失字段（时间范围、默认排序、默认 limit 等）
 - 执行初步安全预检查（是否访问非法实体/字段）

5. PLAN → SQL & Execute（SQL 生成与执行）

- 主要职责：

- 基于 PLAN 和语义层配置生成 SQL（选择语义视图、展开指标与维度、注入过滤与权限）
- 对生成的 SQL 做安全校验和必要重写（只读、limit 等）
- 通过数据库连接池执行 SQL，获得原始结果集
- 不负责自然语言解释，只输出标准化结果数据。

6. Result → Answer（结果解释与回答生成）

- 主要职责：
 - 根据业务需要，将一个或多个子查询的 Result 汇总
 - 结合原始问题、PLAN 摘要与结果数据，生成自然语言回答
 - 处理 Answer 生成失败时的降级（例如仅返回表格和简短说明）

阶段职责原则：

- 每个阶段只关注“自己这一段”的转换，不跨越多个层级混合处理。
- 阶段之间通过明确的数据结构（Context / Subquery / Plan / SqlRequest / Result / Answer）传递信息。
- 如果未来改用不同的 LLM、SQL 生成策略或权限系统，尽量通过替换某个阶段内部实现，而不是改动整条链路。

2.3 流水线编排与并发模型 (Pipeline Orchestration)

为了支持 Stage 1 拆解出的多子查询（Sub-queries）并行处理，系统引入 `PipelineOrchestrator` 组件，位于 API 层与 Stage Controller 之间。

核心职责：

1. **分发 (Dispatch):** 接收 Stage 1 输出的 `QueryRequestDescription`，提取 `sub_queries` 列表。
2. **并发执行 (Concurrency):** 使用 `asyncio.gather` 并行调用 Stage 2 -> Stage 5 的完整链路。每个子查询在一个独立的 `Async Task` 中执行，互不阻塞。
3. **状态隔离 (Isolation):** 为每个子查询创建独立的 `PipelineContext`，确保 `PLAN` 和 `SQL` 生成过程中的变量不发生竞态条件。
4. **结果聚合 (Aggregation):** 等待所有任务完成（或超时），将成功的 `ExecutionResult` 和失败的 `PipelineError` 收集为列表，统一传给 Stage 6 进行最终回答生成。

2.4 设计原则 (Design Principles)

为了使流水线在企业环境中可落地、可运维、可演进，本规范遵循以下设计原则：

1. 可配置 (Configuration-driven)

- 所有业务语义尽量沉淀在语义层 YAML 中，而非硬编码在代码里：

- 域与实体结构
- 指标与维度定义
- 同义词与状态枚举
- 角色与权限范围

◦ 流水线阶段只“消费”这些配置，减少代码修改频率，便于通过配置演进业务。

2. 可观测 (Observability)

- 每个阶段都必须有基础日志和指标，包括：
 - 阶段名称、耗时、输入/输出摘要（必要时脱敏）
 - 成功/失败计数，错误类型分类
- 这样可以对端到端问题进行分段定位：是出在 NL 解析、PLAN 校验还是 SQL 执行。

3. 安全可控 (Security by Design)

- 安全逻辑不是“最后补一刀”，而是在多个阶段嵌入：
 - PLAN 校验阶段进行语义层面的安全预检查
 - PLAN → SQL 阶段注入行级/列级权限
 - Result → Answer 阶段再次过滤敏感字段
- 所有 SQL 必须经过：只读检查、敏感表/字段白名单/黑名单控制、limit 等保护。

4. 可降级 (Graceful Degradation)

- 允许不同阶段在异常情况下采取不同降级策略，例如：
 - NL 解析不确定：返回澄清提示或退化为简单查询
 - SQL 执行超时：返回部分结果或提示缩小时间范围
 - Answer 生成失败：仍返回结构化结果，并给出简单文案
- 避免“一步出错，全链路炸掉”，保证在故障时仍有“可用但不完美”的体验。

5. 可插拔 (Pluggable Components)

- 各阶段实现尽量通过接口（interface/抽象类）隔离：
 - NL→PLAN 可替换不同模型或解析策略
 - PLAN→SQL 可替换不同 SQL builder / 方言支持
 - Answer 生成可替换不同 LLM 或规则模板
- 为后续接入新的模型、数据源、权限系统留出空间，而不推倒重来。

6. 契约驱动开发 (Contract-First Development)

- **核心理念：**本项目采用“SOP 驱动”而非“试错驱动”的开发模式。


```

27 | | — __init__.py
28 | | — stage1_decomposition.py # Subquery 拆解
29 | | — stage2_plan_generation.py # RAG + Prompt -> Plan
30 | | — stage3_validation.py    # 校验与补全 (Validation & Normalization)
31 | | — stage4_sql_gen.py       # Pypika SQL 生成 (Translation & Injection)
32 | | — stage5_execution.py     # SQL 执行与脱敏
33 | | — stage6_answer.py        # 最终回答生成
34 |
35 | — utils/                     # [工具类]
36 | | — __init__.py
37 | | — prompt_templates.py     # 集中管理 Prompt 文本文件
38 | | — log_manager.py          # 结构化日志封装
39 |
40 | — tests/                     # [测试用例]
41 | | — test_registry.py
42 | | — test_stage2.py
43 | | — ...
44 |
45 | — main.py                    # [入口] FastAPI App, 串联 pipeline
46 | — requirements.txt           # 依赖包
47 | — .env                      # [配置] 存放敏感信息: DB_HOST, DB_PASSWORD...
48 | — .env.example               # [规范] 只有 Key 没有 Value, 提交到 Git 供开发参
考

```

考

2.6 技术实现标准 (Technical Implementation Standards)

为了确保系统在高并发场景下的性能表现，并避免依赖冲突，本项目对并发模型与核心技术栈做出以下强制锁定。

2.6.1 异步并发模型 (Asynchronous Concurrency Model)

由于 NL2SQL 链路属于典型的 **I/O 密集型 (I/O Bound)** 场景（主要耗时集中在 LLM 网络请求与数据库查询等待），系统必须严格采用 **全链路异步 (Async/Await)** 模式。

- **入口层 (API Entry):**

- FastAPI 的路由处理函数必须使用 `async def` 定义。
- 严禁在路由中使用同步阻塞操作（如 `time.sleep`, `requests.get`），必须使用 `asyncio.sleep`, `httpx` 等异步替代品。

- **计算层 (LLM & Logic):**

- LLM Client 必须调用 SDK 的异步方法（如 `await client.chat.completions.create(...)` 或 `await chain.ainvoke(...)`）。

- Stage 1 ~ Stage 6 的 Controller 函数签名必须为 `async def`，以支持并发执行子查询拆解等操作。
- 数据层 (Database):
 - 必须使用 SQLAlchemy 的 `AsyncEngine` 和 `AsyncSession`。
 - 底层驱动强制使用异步驱动：PostgreSQL 使用 `asyncpg`，MySQL 使用 `aiomysql` 或 `asyncmy`。
 - Stage 5 的连接获取必须使用 `async with` 上下文管理器。

2.6.2 核心技术栈锁定 (Technology Stack & Dependencies)

为了保证代码的一致性与可维护性，严禁随意引入未经评审的第三方库。核心组件及其版本约束如下：

核心用途 (Purpose)	组件类别	选型技术	版本要求	强制约束 (Constraints)
后端开发语言	Language	Python	3.10+	必须启用严格的 Type Hinting；必须全面使用 <code>asyncio</code> 原生特性。
API 服务框架	Web Framework	FastAPI	0.100+	所有 I/O 操作必须使用 <code>async def</code> ；严禁在路由中写同步阻塞代码。
HTTP 服务器网关	ASGI Server	Uvicorn	Standard	生产环境需配合 Gunicorn (Process Manager) 使用。
数据校验与序列化	Schema Validation	Pydantic	v2.x	严禁使用 v1 。v2 基于 Rust 重写且 API 不兼容，需严格遵循 v2 语法。
(Stage 4) SQL 生成	SQL Builder	PyPika	Latest	仅用于将 Plan 对象翻译为 SQL 字符串；保持纯内存操作，无状态。
(Stage 5) SQL 执行	DB Driver & Pool	SQLAlchemy	2.0+	仅使用其 <code>AsyncEngine</code> 做连接池管理和 Raw SQL 执行； 严禁使用其 ORM 模型定义和查询语法 。
大模型 API 调用	LLM SDK	OpenAI / LangChain	Latest	优先使用官方原生 SDK 以降低延迟；必须使用异步调用方法 (<code>await</code>)。
向量数据库	Vector DB	Qdrant	Latest	必须使用 <code>qdrant-client</code> ； 必须使用 <code>AsyncQdrantClient</code> 以保持全链路异步特性。
Embedding 服务	AI Model	Jina AI	v3 (API)	用于语义向量生成； 必须通过 <code>httpx (Async)</code> 调用 REST API ，严禁使用同步 <code>requests</code> 库。
环境变加载	Environment	python-dotenv	Latest	仅在 <code>main.py</code> 或 <code>db_connector.py</code> 初始化时使用，禁止在业务逻辑中重复读取。

3. 各阶段详细设计 (Pipeline Stages)

本章按照流水线的时序，对 6 个阶段逐一展开。

3.1 阶段一：subquery 拆解与上下文建模（Stage 1: Subquery Decomposition & Context）

3.1.1 阶段目标与职责

本阶段的目标是：

接收用户的原始自然语言问题和基础上下文字段，通过 LLM 将问题拆解为结构化的 sub_queries，同时构建统一的 RequestContext，输出一个标准的「查询请求描述」结构体对象。

核心职责：

- 接收调用方传入的：
 - 原始自然语言问题（question）
 - 基础用户上下文（user_id / role_id / tenant_id / locale）
- 在后端补全必要的上下文字段（如 request_id、默认 timezone 等），构建 RequestContext
- 使用 LLM 将原始问题拆解为一个或多个 **对 SQL 友好的 subquery 描述**
- 将 `sub_queries + RequestContext` 组合为统一的内部结构，供后续 NL→PLAN 阶段使用

本阶段不做：

- 不解析 metrics / dimensions / time_range 等细节（NL→PLAN 的职责）
- 不生成 PLAN / SQL
- 不做权限注入（在 PLAN→SQL 阶段完成）

3.1.2 输入与输出（Stage I/O）

输入（逻辑层面）

调用方在 API 请求中至少需要提供以下字段：

- question: string, 原始自然语言问题（必填）
- user_id: string, 用户 ID（必填，用于权限判定）
- role_id: string, 角色标识（必填，用于行/列级权限）
- tenant_id: string, 租户标识（必填，用于多租户隔离）
- locale: string, 语言区域（必填，如 zh-CN / en-US，用于 Answer 语言与格式）
- current_date: string 必填，LLM 依赖它做时间推断

后端补全字段：

- `request_id`：每次请求唯一 ID

输出（内部对象：「查询请求描述」对象,代码中定为：QueryRequestDescription 对象）

本阶段输出一个统一的内部结构（示意）：

代码块

```
1  {
2    "request_context": {
3      "request_id": "req_20250310_00001"
4      "user_id": "u_123",
5      "role_id": "SALES_MANAGER",
6      "tenant_id": "t_demo",
7      "locale": "zh-CN",
8      "current_date": "2023-11-02", // 必填，LLM 依赖它做时间推断
9    },
10   "sub_queries": [
11     {
12       "id": "req_20250310_0001-00001", //是request_id后面再拼了一个顺序号
13       "description": "统计每个区域的销售额" //是对原始用户问题的拆分
14     }
15     {
16       "id": "req_20250310_0001-00002", //是request_id后面再拼了一个顺序号
17       "description": "取销售额中的前5名" //是对原始用户问题的拆分
18     }
19     // 可能还有 sub_query3、sub_query4 ...
20   ]
21 }
```

- `request_context`：本阶段构建好的标准上下文对象
- `sub_queries`：由 LLM 拆解生成的子查询列表，后续 NL→PLAN 将针对每个 subquery 生成 PLAN

3.1.3 处理过程

从本阶段开始，所有后续阶段不再直接访问 HTTP 请求，而是统一使用 `request_context` 对象；

- 日志、监控、SQL 执行信息等均需带上 `request_id`

具体过程如下5步：

第一步：构造 RequestContext

在进入 LLM 之前，先根据 HTTP 请求构建 `RequestContext`：

- 从请求中读取：`user_id` / `role_id` / `tenant_id` / `locale` / `current_date`
- 在后端生成：`request_id` (`request_id = f"req_{datetime.now().strftime('%Y%m%d%H%M%S')}-{uuid.uuid4().hex[:8]}"`)

- 得到request_context

第二步：调用 LLM 得到 sub_queries

提示词代号：PROMPT_SUBQUERY_DECOMPOSITION，详情请查看附录A

第三步：后验规则最终校验

如果 sub_queries 数量为 0或> 3：视为 LLM 输出不合法，抛出异常(使用本项目自定义的Error类，包含：request_id，error_code =SUBQUERY_JSON_INVALID, error_stage=stage1,message= “LLM拆解的自然语言数量不对”)

如果 sub_queries 数量为 1且description = INVALID_QUERY：视为用户问题不合法，抛出异常(使用本项目自定义的Error类，包含：request_id，error_code =INVALID_QUERY, error_stage=stage1,message= “无法理解用户输入”)

第三步：生成内部用的 sub_query ID

拿到 LLM 输出（字符串），解析JSON字符串，变成subquery_list

subquery_list 必须是一个非空 list；

每个元素都必须有非空的 description 字段；

每个subquery_id规则：id = f"{request_id}-{序号}"，序号用零填充，比如 00001、00002。

伪代码：

代码块

```
1 normalized_subqueries = []
2
3 for idx, sq in enumerate(subquery_list, start=1):
4     new_id = f"{request_id}-{idx:05d}" # 00001 这种格式
5     normalized_subqueries.append({"id": new_id,"description":
6         sq["description"].strip()
7     })
```

- LLM 给的 id (q1/q2) 可以直接忽略
- 对外统一使用 req_xxx-00001 这种 ID，更稳定可控。

第五步：组合成 QueryRequestDescription 对象

伪代码：

```
1  QueryRequestDescription = {  
2    "request_context": request_context,  
3    "sub_queries": subquery_list  
4  }
```

到这里为止，Stage 1 的输出就是一个完整的 `QueryRequestDescription` 对象，阶段二（NL→PLAN）只要遍历 `qrd.sub_queries` 去做 PLAN 解析就行了。

3.2 阶段二：NL → PLAN 阶段（Stage 2: NL → PLAN）

3.2.1 阶段目标与职责

本阶段是整个流水线的“大脑”。其核心职责是接收 Stage 1 拆解出的自然语言子查询（`sub_query`），结合业务语义定义（Schema），推断用户的分析意图，并生成一个结构化的、与数据库方言无关的 **查询计划（PLAN）**。

核心原则：

- **语义对齐**：将用户口语（如“营收”）准确映射为系统内部 ID（如 `metric_revenue`）。
- **去幻觉**：严禁 LLM 编造不存在的指标或维度 ID。
- **意图结构化**：将自然语言转化为计算机可理解的 JSON 对象（指标、维度、筛选、排序）。

3.2.2 输入与输出

输入（Input）：

- `sub_query`：单条子查询对象（来自 Stage 1 输出列表中的一项，由 Orchestrator 分发）。
- `request_context`：用户上下文（包含当前时间、用户角色、租户 ID）。
- `SemanticRegistry`：语义层注册表（包含所有指标、维度、实体的定义与向量索引）。

输出（Output）：

- `Plan` 对象：一个符合 V1 简化规范的 JSON 对象。
 - **关键差异说明**：本阶段输出的 Plan 不包含 SQL 语法细节（如 `HAVING`、`JOIN`），只包含业务意图。

3.2.3 处理流程（Process Flow）

本阶段不再使用传统的 NLP 分词或槽位填充技术，而是采用 **RAG（检索增强生成）** 模式，具体分为四个步骤：

准备工作：语义索引构建与冷启动优化 (Semantic Indexing & Cold-Start Optimization)

服务启动时，`SemanticRegistry` 单例负责加载 YAML 配置并构建向量索引，把每个指标/维度的“名字+描述”转成向量存好。

为了平衡“启动速度”与“Token 成本”，系统采用“配置指纹 (Config Fingerprint)”机制，结合 **Qdrant** 向量数据库进行管理。

执行逻辑：

- 1. 指纹计算：** 读取 `semantic_metrics.yaml` 和 `semantic_core.yaml` 文件内容，计算 MD5 哈希值（即 `current_fingerprint`）。
- 2. 状态检查：** 查询 Qdrant 中的系统状态集合（如 `sys_config_state`），获取 `last_fingerprint`。
- 3. 分支处理：**
 - **Case A (无变更 - Fast Path):** 若 `current_fingerprint == last_fingerprint`，跳过 Embedding 过程，直接复用 Qdrant 中现有的向量数据。（启动耗时 < 1s）
 - **Case B (有变更 - Re-index Path):** 若指纹不一致：
 - a. 清空 Qdrant 中的业务集合。
 - b. 调用 Jina AI API 对 YAML 中的所有 Term (Metric/Dimension/Enum) 进行 Embedding。
 - c. 将向量批量 Upsert 入 Qdrant。
 - d. 更新 `sys_config_state` 中的指纹为 `current_fingerprint`。
- 4. 运行时：** Stage 2 的检索逻辑直接调用 Qdrant 的 `search` 接口。（关于 SemanticRegistry 的完整定义，请看本文末尾：附录F 关键类和算法说明）

示例逻辑伪代码如下：

代码块

```
1  class SemanticRegistry:
2      def __init__(self):
3          # 1. 倒排索引：用于“精确匹配”
4          # 结构：{ "别名/名称": ["METRIC_GMV"] }
5          # 作用：用户搜“销售额”，直接找到 METRIC_GMV
6          self.keyword_index: Dict[str, List[str]] = {}
7
8          # 2. 元数据字典：用于“查详情”
9          # 结构：{ "METRIC_GMV": { ...详细定义... } }
10         self.metadata_map: Dict[str, SchemaDef] = {}
11
12         # 3. [变更] 向量客户端 (Qdrant)：用于“向量搜索”
13         # 原 self.vector_index (List) 已移除，改为托管给 Qdrant
14         self.qdrant_client = QdrantClient(host=..., port=...)
15
16     def load_from_yaml(self):
```

```

17         """
18         启动时执行的核心加载逻辑 (Cold-Start Optimization)
19         """
20         # 1. 计算指纹
21         current_fingerprint = calculate_md5("semantics/*.yaml")
22         last_fingerprint = self.qdrant_client.get_sys_state("fingerprint")
23
24         # 2. 加载元数据 (Metadata) -> 填充 metadata_map & keyword_index
25         # 这一步很快, 总是执行
26         self._load_metadata_into_memory()
27
28         # 3. 检查向量索引状态
29         if current_fingerprint == last_fingerprint:
30             logger.info("配置无变更, 复用 Qdrant 现有向量索引 (Fast Path)")
31             return
32
33         # 4. 重建索引 (Re-index Path)
34         logger.info("配置变更, 开始重建向量索引...")
35         # a. 调用 Jina AI 生成 Embedding
36         # b. 写入 Qdrant (注意: Payload 必须包含 id)
37         points = []
38         for term in self.metadata_map.values():
39             vector = self.jina_client.encode(term.description)
40             points.append(PointStruct(
41                 id=uuid.uuid4(),
42                 vector=vector,
43                 payload={"id": term.id} # <--- 关键: 存入 ID 以便检索时取回
44             ))
45         self.qdrant_client.upsert(collection_name="schema_collection",
46 points=points)
47
48         # c. 更新指纹
49         self.qdrant_client.set_sys_state("fingerprint", current_fingerprint)

```

其中:

代码块

```

1  # self.metadata_map
2  {
3      "METRIC_GMV": {
4          "id": "METRIC_GMV",
5          "name": "GMV",
6          "domain_id": "SALES",      # <--- stage2权限过滤核心判断字段
7          "type": "METRIC",
8          "aliases": ["销售额", "营收"],

```

```

9      ...
10    },
11    "DIM_CITY": {
12      "id": "DIM_CITY",
13      "name": "城市",
14      "domain_id": "SALES",
15      "type": "DIMENSION",
16      ...
17    }
18    # ... 几百个这样的键值对 ...
19  }

```

代码块

```

1  # self.keyword_index
2  {
3      "gmV": ["METRIC_GMV"],
4      "销售额": ["METRIC_GMV"],
5      "营收": ["METRIC_GMV"],
6      "城市": ["DIM_CITY"],
7      "地区": ["DIM_CITY"]
8  }

```

代码块

```

1  # Qdrant Collection: "schema_collection" (数据库)
2  # Point Structure:
3  {
4      "id": "UUID...",
5      "vector": [0.12, -0.45, ...], # Jina Embedding
6      "payload": {
7          "id": "METRIC_GMV"          # <--- 检索时只返回这个 ID
8      }
9  }

```

步骤一：权限过滤与混合检索 (Security-First Hybrid Retrieval)

目的：在确保数据安全的前提下，精准召回相关 Schema。

1. 权限预过滤 (Security Filter):

- 根据 request_context.user_id / role_id，从 SemanticRegistry 中获取该用户**可见**的指标与维度白名单。（注意：严禁将用户无权的字段喂给 LLM）

白名单的构造过程如下：

- i. **输入：** `role_id` (例如 "ROLE_SALES_STAFF")。
- ii. **读取配置：** 读取 `semantic_security.yaml` 中的 `role_policies`。
- iii. **获取域权限：** 获取该角色的 `domain_access` 列表 (如 `["SALES"]`)。
- iv. **注入公共域 (Critical Fix)：强制追加 "COMMON" 域。**
逻辑： `allowed_domains = set(domain_access) | {"COMMON"}`。
理由：确保通用维度（如时间、地理位置）对所有用户可见，否则无法进行基础分析。
- v. **筛选 ID：**
遍历 `SemanticRegistry.metadata_map`：
若 `term.domain_id` 在 `allowed_domains` 中 -> **保留**。否则 -> **丢弃**。
- vi. **输出结果如下：**

代码块

```
1 allowed_ids = ["METRIC_GMV", "METRIC_ORDER_CNT", "DIM_CITY", ...]
2 # 这个 List[str] 就是下一步【B.向量检索】中用于过滤候选池的唯一依据
```

2. 混合召回 (Hybrid Search):

- **A. 精确匹配：** 检查用户问题中是否包含 Schema 中的 `name` 或 `aliases`。若命中，强制保留该候选项。
 - 这里不切分 `sub_query`，而是拿着 SemanticRegistry 里存的“别名表”去 `sub_query` 里找。算法逻辑（伪代码）如下：

代码块

```
1 user_query = "营收怎么样"
2 matched_ids = []
3
4 # 遍历所有已知的别名 (数量有限, 几千个而已, 速度极快)
5 for alias, target_id in registry.keyword_index.items():
6     if alias in user_query: # 核心就是这一句: 字符串包含判断
7         matched_ids.append(target_id)
8
9 # 结果: ["METRIC_GMV"]
```

◦ B. 向量检索:

将 `sub_query` 转化为向量，调用 Qdrant 的 Search 接口。

关键改进: 权限过滤不再是取出所有向量在内存里 `if` 判断，而是直接转化为 Qdrant 的 **Filter (Payload Filtering)**，利用向量数据库的索引能力进行**预过滤 (Pre-filtering)**，性能显著提升。

算法逻辑伪代码:

代码块

```
1  # --- Step 2B: 向量检索 (Qdrant Implementation) ---
2
3  # 1. 向量化 (Embedding)
4  # 调用 Jina AI 将用户自然语言转为向量
5  query_vector = jina_client.encode(user_query)
6
7  # 2. 构造权限过滤器 (Payload Filter)
8  # 将 Step 1 产出的 allowed_ids (白名单) 转化为 Qdrant 的过滤条件
9  # 逻辑: 只检索 id 在 allowed_ids 列表中的记录
10 search_filter = Filter(
11     must=[
12         FieldCondition(
13             key="id",
14             match=MatchAny(any=allowed_ids) # 对应 Qdrant 的 "in" 逻辑
15         )
16     ]
17 )
18
19 # 3. 执行检索 (Search)
20 # 直接获取 Top-N 结果, 无需手动计算点积 (dot product)
21 search_results = self.qdrant_client.search(
22     collection_name="schema_collection",
23     query_vector=query_vector,
24     query_filter=search_filter,
25     limit=30 # 对应调参锚点 vector_search_top_k
26 )
27
28 # 4. 格式转换
29 # 将 Qdrant 返回的 ScoredPoint 转换为内部使用的 (id, score) 元组
30 valid_results = []
31 for hit in search_results:
32     # 阈值检查 (Similarity Threshold) 可在这里做, 也可以在后续统一做
33     valid_results.append((hit.payload["id"], hit.score))
34
35 # 结果: valid_results 已经是排序好的 Top-N
```

注意：这里有**调参锚点1(vector_search_top_k)**，定义：上述向量检索的Top-N，暂定30

为了方便后续调参（Tuning）并保持代码硬编码最小化，本项目的关键超参数统一托管在全局配置类(PipelineConfig)中,, 具体定义和有哪些配置，请看本文末尾【附录D】

3. 合并与截断：

- 将 A (精确匹配) 和 B (向量搜索) 的结果集合并，再移除重复的 ID。
- **截断策略：**

核心： 优先保障精确匹配，剩余名额由高分向量填充，且必须满足相似度门槛。

i. 确定容量：

- 读取全局配置 pipeline_config.max_term_recall (**调参锚点2(max_term_recall)**，定义：最终喂给LLM的term数量上限，暂定20)。

ii. VIP 入列 (A类)：

- 将所有 **精确匹配** 的结果加入最终列表 final_list。 (注：精确匹配通常数量很少，一般 1-3 个)

iii. 向量填空 (B类)：

- 计算剩余名额：remaining_slots = 20 - len(final_list)。
- 如果 remaining_slots <= 0，直接结束。
- 从向量检索结果中，**按分数降序遍历**：
 - **去重检查**：如果该 ID 已经在 final_list 中，跳过。
 - **阈值检查**：如果分数 < pipeline_config.similarity_threshold (**调参锚点3(similarity_threshold)**，定义：之前向量检索阶段score的阈值，暂定0.4)，停止填充（宁缺毋滥，不要凑数）。
 - **入列**：加入 final_list。
 - **满员检查**：一旦填满 remaining_slots，停止。

4. 上下文组装：

目的： 将步骤三筛选出的最终候选列表 (final_list), 序列化为 LLM 易读、高密度的文本格式 (Schema Context) ，准备注入 Prompt。由 Stage 2 主流程调用辅助函数 `format_schema_context(final_list, registry)` 执行。

输入数据 (Input)：

- `final_list`：步骤 3 输出的 Term 对象列表（内存对象）。
- `registry`：SemanticRegistry 实例（用于反查枚举定义）。

处理逻辑 (Logic Flow)：

步骤 A：分类 (Grouping)

- 将 `final_list` 中的 Term 对象按类型拆分为两组： `[METRICS]` 和 `[DIMENSIONS]`。
- **排序**：在组装前，按 `term.id` 字母序对列表进行排序，保证 Prompt 的确定性，避免因顺序抖动导致 LLM 输出不稳定。

步骤 B：单行格式化 (Line Formatting)

函数遍历每个 Term 对象，读取其内存属性生成描述字符串，生成单行描述：

a. 基础字段（通用）：

- 模板： `- ID: {id} | Name: {name} | Aliases: {alias_str} | Desc: {desc}`
- 其中的防御性追加：
 - **Aliases**：读取 **当前 Term 对象** 的 `aliases` 属性。若非空，追加 `| Aliases: a, b`。若为空，**直接不显示**（节省 Token）。
 - **Desc**：读取 **当前 Term 对象** 的 `description` 属性。若非空，追加 `| Desc: {description}`。对过长的描述(超过50个字符)截断（**调参锚点 4(max_description_length)**，定义：**Schema Context**中描述的最大字符数 (中文字符)，暂定50）。

b. 类型特有逻辑（关键）：

- **Metrics**：仅展示基础字段。
- **Dimensions**：
 - **时间标记**：若 `is_time=True`，追加 `| Is_Time: True`。提示 LLM 该字段可用于 `time_grain` 分组。
 - **枚举透出 (Enums Exposure)**：
 - 读取对象属性 `term.enum_ref`，若非空，调用 `registry.get_enum_def(enum_ref)` 获取枚举定义对象。
 - 读取枚举对象的 `values` 列表，截取前8个（**调参锚点 5(max_enum_values_display)**，定义：**Schema Context**中，单个维度最多展示多少个枚举值示例，暂定8）。追加 `| Values: [A, B, C, ...]`。
 - **作用**：提示 LLM 该字段的合法取值范围，防止生成幻觉值（如将“已发货”编造为 "Sent" 而非 "Shipped"）。

步骤 C：拼接 (Joining)

- 将格式化后的行，加上 `[METRICS]` 和 `[DIMENSIONS]` 标题，拼接成最终的大字符串。
- 增加防御逻辑：`if len(enum_values) > 50: values_display = []`（超过50个就不透出给 LLM，防止 Context 爆炸）

最终输出示例 (Schema Context)：

- **名称：Schema Context**
- **定义：**一个经过序列化、截断和格式化的纯文本字符串，包含了本次查询所需的指标与维度定义。
- **用途：**将直接替换 Prompt 模板中的 `{schema_context}` 占位符。

代码块

```
1  [METRICS]
2  - ID: METRIC_GMV | Name: GMV (收入口径) | Aliases: 销售额, 营收 | Desc: 订单总金额
3  - ID: METRIC_ORDER_CNT | Name: 订单数 | Aliases: 订单量 | Desc: 有效订单计数
4
5  [DIMENSIONS]
6  - ID: DIM_COUNTRY | Name: 国家 | Aliases: 地区 | Desc: 订单发生国家
7  - ID: DIM_ORDER_STATUS | Name: 订单状态 | Values: [Resolved, Shipped, In
    Process, On Hold]
8  - ID: DIM_ORDER_DATE | Name: 订单日期 | Aliases: 下单时间 | Is_Time: True
```

步骤二：Prompt 构建与 LLM 生成 (Prompt Engineering & Generation)

目的：

将步骤一检索到的 Schema 上下文与用户问题组装为标准 Prompt，并调用 LLM 生成符合规范的 Query Plan JSON。

1. 消息构建 (Message Construction)

代码将 Prompt 模板拆分为 System 和 User 两个部分，并注入动态变量：

- **System Message（系统指令）：**
 - **角色设定：**定义 LLM 为“数据分析专家”。
 - **输出约束：**强制要求输出符合定义的 JSON 结构（Plan Schema）。
 - **逻辑规则：**包含 Intent 分类、同环比判断、时间粒度处理等核心业务规则。
- **User Message（用户输入）：**
 - **当前环境 ({current_date})：**注入当前日期（格式 YYYY-MM-DD），用于推断“今年”、“上月”等相对时间。
 - **可用词表 ({schema_context})：**注入由步骤一生成的 Schema 文本卡片。
 - **约束：**明确告知 LLM 仅允许使用列表中的 ID，严禁自创。
 - **用户问题 ({user_query})：**用户的原始自然语言提问。

2. 执行策略 (Execution Strategy)

调用 LLM API 时，必须应用以下参数配置以保证稳定性：

- **确定性控制：**设置 `temperature = 0`，消除随机性，保证相同输入得到相同输出。

- **格式强制：**设置 `response_format = { "type": "json_object" }`，确保输出为合法 JSON，减少解析错误。
- **具体提示词代号：** `PROMPT_PLAN_GENERATION`，详情请看【附录A】

步骤三：解析与反幻觉清洗 (Parsing & Anti-Hallucination)

目的：将不稳定的LLM输出的文本转换为干净的 Plan 对象（初步 Plan（Skeleton Plan）），供 Stage 3 进行深度校验。

1. JSON 提取：

- 检查并去除LLM输出文本中的 Markdown 标记（````json ... ````）。
- `json.loads` 解析为字典。

2. 结构化实例化：

- 利用 Pydantic 模型（`Plan`）进行实例化。
- 作用：自动校验字段类型（如 `limit` 必须是 int），若失败直接抛错或重试。

3. 幻觉 ID 剔除 (Pruning)：

- 遍历：`metrics`，`dimensions`，`filters`，`order_by` 中的所有 `id`。
- 验证：调用 `SemanticRegistry.validate_id(id)`。
- 清洗：若 ID 不存在，物理删除该条目，并写入 `Plan.warnings`。
- 注意：这里只管“ID 存不存在”，不管“ID 之间兼不兼容”（那是 Stage 3 的事）。

4. 空值熔断：

- 若清洗后 `metrics` 和 `dimensions` 均为空，抛出 `EMPTY_PLAN_ERROR`，提前终止。

3.3 阶段三：PLAN 校验与标准化 (Stage 3: PLAN Validation & Normalization)

3.3.1 阶段目标与职责

本阶段承接 Stage 2 输出的初步 Plan（Skeleton Plan），负责执行深度的语义逻辑校验与业务规则补全，确保最终生成的 Plan 是逻辑自治、业务合规且具备执行条件的。

核心原则：

- **逻辑自治：** 确保指标与维度在业务上可以组合（防止笛卡尔积）。

- **业务补全：** 自动填充默认时间窗口、默认 Limit 等兜底逻辑。
- **单事实表约束：** MVP 阶段严格限制查询必须限定在单一业务实体内。

3.3.2 输入与输出

输入 (Input)：

- Skeleton Plan：来自 Stage 2 的初步 Plan 对象（已通过基础 JSON/ID 检查）。
- SemanticRegistry：语义层注册表（提供指标/维度的元数据与关系图谱）。
- request_context：用户上下文（用于权限复核）。

输出 (Output)：

- Validated Plan：完全合规、补全默认值后的 Plan 对象。
- **异常处理：** 若校验失败，抛出具体的业务异常（如 SemanticError），拒绝生成 SQL。

3.3.3 处理流程 (Process Flow)

本阶段采用“流水线检查点 (Checkpoints)”模式，按顺序执行以下 4 个步骤：

步骤一：结构完整性与基础清洗 (Structural Sanity)

目的： 确保 Plan 对象的基础结构合法，不是“缺胳膊少腿”的残次品。

1. 必填字段与意图合法性检查 (Mandatory Field & Intent Check)

- **目的：**
确保 Plan 的核心骨架完整，防止出现“无法执行”的逻辑空洞（如：要聚合统计却没给指标）。
- **执行过程：**
 - **意图校验：** 检查 `Skeleton Plan.intent` 是否属于允许的枚举集合 `{"AGG", "TREND", "DETAIL"}`。
 - **指标依赖校验：** 若 `intent` 为 `AGG` 或 `TREND`，检查 `Skeleton Plan.metrics` 列表是否非空。
- **失败/降级策略：**
 - **情形 A：意图非法 (Invalid Intent)**
 - **错误分类：** 属于 **第三级：错误 (Error - System Failure)**。
 - **流转动作：** 抛出 `INVALID_PLAN_STRUCTURE` 异常，熔断流程。这通常意味着 Stage 2 的 LLM 生成严重失控，需检查 Prompt 或模型状态。
 - **情形 B：缺少指标 (Missing Metrics)**
 - **错误分类：** 属于 **第二级：澄清 (Clarification)**。
 - **流转动作：** 抛出 `MISSING_METRIC_ERROR` 异常，熔断流程。

- **Stage 6 响应：**捕获异常并追问用户：“请问您具体想统计哪些指标？（如：销售额、订单量等）”。

2. 列表字段初始化 (List Field Initialization)

- **目的：**
消除 `None` 值隐患，确保后续步骤（如遍历 `filters`）时无需反复判空，简化代码逻辑。
- **执行过程：**
 - **扫描字段：**检查 `filters` , `order_by` , `dimensions` , `warnings` 等列表类型字段。
 - **空值清洗：**若字段值为 `None` , 将其强制赋值为 `[]` （空列表）。
- **失败/降级策略：**
 - **策略：自动修正 (Auto-Correction)。**
 - **动作：**静默修复，不报错，不记录 Warning。这是系统内部的数据标准化过程，对用户透明。

步骤二：术语存在性与权限复核 (Term Existence & Access)

目的： 作为 Stage 2 的“双重保险”，防止漏网的幻觉 ID 或越权访问。

1. ID 存在性复核 (Existence Verification)

- **目的：**
作为 Stage 2 的“双重保险”，防止 LLM 编造的幻觉 ID（Stage 2 未清洗干净的漏网之鱼）进入后续流程，导致 SQL 生成报错。
- **执行过程：**
 - **遍历检查：**代码分别遍历 Skeleton Plan 的 `metrics` 列表和 `dimensions` 列表。
 - **调用验证：**针对每个元素的 `id` , 调用 `SemanticRegistry.get_metric(id)` 或 `SemanticRegistry.get_dimension(id)` 。
 - **判定逻辑：**若 Registry 返回 `None` , 则判定该 ID 为非法幻觉。
- **失败/降级策略：**
 - **错误分类：**属于 **第一级：警告 (Warning - Partial Invalidity)**。
 - **流转动作：**
 - Stage 3 (修正)：**从 Skeleton Plan 中**物理移除**该不存在的指标或维度对象。
 - Stage 3 (记录)：**向 `Skeleton Plan.warnings` 列表追加告警信息（如 `"指标 'METRIC_UNKNOWN' 不存在，已自动忽略"`）。
 - Pipeline (继续)：**继续执行后续流程。
 - 例外熔断：**若移除后 `metrics` 列表为空（且意图为聚合），则升级为 **第二级：澄清**，抛出异常中断流程（因为查不到任何数据了）。

2. Filter 字段合法性校验 (Filter Field Validation)

- **目的：**
确保过滤条件作用于真实存在的字段，防止生成 `WHERE unknown_col = 'val'` 这样的无效 SQL。
- **执行过程：**
 - **遍历检查：**遍历 Skeleton Plan 的 `filters` 列表。
 - **调用验证：**提取每个 filter 的 `id`，检查该 ID 是否存在于 `SemanticRegistry` 中（即 `get_metric(id)` 或 `get_dimension(id)` 有返回值）。
 - **判定逻辑：**若 ID 查无此人，或 ID 存在但类型不符（如引用了纯实体 ID 而非维度 ID），判定为非法 Filter。
- **失败/降级策略：**
 - **错误分类：**属于 **第一级：警告 (Warning - Partial Invalidity)**。
 - **流转动作：**
 - Stage 3 (修正)：**从 Skeleton Plan 的 `filters` 列表中**物理移除**该非法 Filter。
 - Stage 3 (记录)：**向 `Skeleton Plan.warnings` 列表追加告警信息（如 `"过滤条件字段 'DIM_FAKE' 不存在，已自动忽略该条件"`）。
 - Pipeline (继续)：**继续执行后续流程（视为无该条件的宽泛查询）。
 - Stage 6 (响应)：**LLM 在生成回答时，通过读取 `warnings` 提示用户：“注意：部分过滤条件因字段无法识别已失效。”

3. 权限复核 (Permission Enforcement) —— 必须执行

- **目的：**
防御 **Prompt 注入攻击**。防止恶意用户通过诱导性 Prompt 绕过 Stage 2 的检索限制，强行让 LLM 输出了用户无权访问的敏感 ID（如 `METRIC_CEO_SALARY`）。
- **执行过程：**
 - **获取上下文：**读取 `request_context.role_id`。
 - **获取白名单：**调用 `SemanticRegistry.get_allowed_ids(role_id)`，获取该角色授权访问的全量 ID 集合。
 - **收集全量 ID：**扫描 Skeleton Plan 中所有字段（`metrics`，`dimensions`，`filters`，`order_by`）中引用的所有 ID。
 - **集合比对：**判断 **[Plan ID 集合]** 是否完全属于 **[权限白名单]** 的子集。
- **失败/降级策略：**
 - **错误分类：**属于 **第三级：错误 (Error - Security Violation)**。
 - **流转动作：**

- i. **Stage 3 (抛出)**: 直接抛出 `PERMISSION_DENIED` 异常。**严禁**做“剔除后继续”的降级处理（因为这是安全事件，不是数据质量问题）。
- ii. **Pipeline (熔断)**: **立即终止**后续流程，**跳过** Stage 4 和 Stage 5。
- iii. **Stage 6 (响应)**: 捕获异常，返回标准化的权限拒绝提示：“您无权访问查询中涉及的部分数据（如：薪资、成本等），请联系管理员。”

步骤三：语义连通性校验 (Semantic Connectivity) —— 核心步骤

目的：确保指标和维度能“玩到一起”，防止生成无意义的查询。

1. Metric-Dimension 兼容性 (Metric-Dimension Compatibility)

- **目的**：
确保指标 (Metric) 在物理模型上确实能按指定的维度 (Dimension) 进行分组统计，避免生成产生笛卡尔积或无意义数据的 SQL
- **执行过程**:
 - **遍历检查**：代码双重遍历 Skeleton Plan 的 `metrics` 列表和 `dimensions` 列表
 - **调用验证**：针对每一对 `(metric_id, dimension_id)`，调用 `SemanticRegistry.check_compatibility(metric_id, dimension_id)`。
 - **判断逻辑**：Registry 内部检查指标所属的主表 (Entity) 是否拥有指向该维度的外键，或是否在 `semantic_core.yaml` 的 `relationships` 中存在关联路径。
 - **场景示例**:
 - 用户查询：“按**客户**统计**库存量**”。
 - Plan: Metric=`METRIC_INVENTORY` (主表: 库存表), Dimension=`DIM_CUSTOMER` (主表: 客户表)。
 - 校验：库存表没有 `customer_id` 外键（库存还在仓库里，没卖给客户）。
 - 结果：`check_compatibility` 返回 `False`。
- **失败/降级策略**:
 - **策略**：降级 (Degrade)，属于 **第一级：警告 (Warning - Partial Invalidity)**。
 - **动作**:
 - i. **Stage 3 (修正)**：从 Skeleton Plan 的 `dimensions` 列表中**物理移除**该不兼容维度（如 `DIM_CUSTOMER`）。
 - ii. **Stage 3 (记录)**：向 `Skeleton Plan.warnings` 列表追加告警信息（保留原始语句）。
 - iii. **Pipeline (继续)**：**继续执行**后续流程（进入 Stage 4 生成 SQL）。此时 Plan 已变更为“无分组”的聚合查询（查总库存量）。
 - iv. **Stage 6 (响应)**：

- **展示：** LLM 在生成最终回答时，读取 `warnings` 字段。**话术示例：**“当前库存总量为 10,000 件。（注：维度 '客户' 无法用于统计 '库存量'，已自动忽略该分组条件。）”

2. Metric-Metric 兼容性 (Metric-Metric Compatibility)

- **目的：**

在 MVP 阶段严格限制**单事实表查询**，防止出现跨事实表（Drill-across）的复杂 JOIN 逻辑，确保数据准确性。

- **执行过程：**

- **获取归属：** 遍历 Skeleton Plan 的 `metrics` 列表，调用 `SemanticRegistry.get_metric(id)` 获取每个指标定义的 `entity_id` 属性。
- **一致性比对：** 检查所有指标的 `entity_id` 是否完全一致。
- **场景示例：**
 - 用户查询：“看下**销售额**和**库存量**”。
 - Plan: Metric A= `METRIC_SALES` (Entity: `SALES_ORDER`), Metric B= `METRIC_INVENTORY` (Entity: `INVENTORY_FACT`)。
 - 比对: `SALES_ORDER` != `INVENTORY_FACT` 。
 - 结果：判定为跨事实表查询。

- **失败/降级策略：**

- **策略：** 熔断（Fail Fast），属于 **第三级：错误 (Error - Capability Boundary)**。
- **动作：**
 - i. **Stage 3 (抛出)：** 直接抛出 `UNSUPPORTED_MULTI_FACT` 异常，携带冲突的实体名称（如 `["销售域", "库存域"]`）。
 - ii. **Pipeline (熔断)：** **立即终止**后续流程，**跳过** Stage 4 (SQL生成) 和 Stage 5 (SQL执行)。
 - iii. **Stage 6 (捕获与响应)：**
 - **捕获：** 在全局异常处理器中捕获该异常。
 - **响应生成：** 将技术异常转换为友好的业务提示，明确告知用户系统限制。**话术示例：**“抱歉，目前暂不支持同时查询‘销售’和‘库存’两个不同业务域的数据。建议您拆分为两个问题分别提问。”
- **理由：** 跨表查询涉及复杂的 Full Outer Join 和粒度对齐，MVP 阶段不支持“瞎猜”或“硬连”，必须明确拒绝并提示用户分两次查询。

步骤四：标准化与业务规则注入 (Normalization & Injection)

目的： 补全默认值，注入强制规则，让 Plan 变得“丰满”。

1. 时间窗口补全 (Time Window Injection)

- **目的：**

防止用户未指定时间范围时查询全量历史数据，导致数据库性能崩溃或业务口径错误（如“销售额”通常默认看当月）。

- **执行过程：**

- **前置检查：** 检查 `Skeleton Plan.time_range`。若不为空（用户已显式指定），直接跳过此步。
- **主指标锚定：**
 - 取 `Skeleton Plan.metrics[0]`，即：列表的**第一个元素**作为“主指标”（Primary Metric）。
 - 从 `SemanticRegistry` 读取主指标的 `default_time` 配置。
- **冲突检测：**
 - 若主指标有默认时间，将其暂定为全局时间。
 - 遍历剩余指标，检查其 `default_time` 是否与主指标一致。若不一致，标记为“隐式冲突”，但不阻断流程。

- **失败/降级策略：**

- **情形 A：成功补全（含潜在冲突）**
 - **策略：** 自动修正 (Auto-Correction)。属于 **第一级：警告 (Warning)**。
 - **动作：**
 - **Stage 3 (注入)：** 将主指标的默认时间解析并填入 `time_range` 字段。
 - **Stage 3 (告警)：**
 - 若无冲突：追加提示 “未指定时间，已按主指标默认展示最近 30 天数据”。
 - 若有冲突：追加提示 “未指定时间，已优先采用主指标 '{metric_name}' 的默认时间（最近 30 天），忽略了 '{other_metric}' 的默认设置”。
 - **Stage 6：** 在生成最终回答时，将此 Warning 展示给用户，确保业务口径透明。
- **情形 B：主指标也无默认值**
 - **策略：** 系统兜底 (System Fallback)。
 - **动作：**
 - **Stage 3：读取**
`SemanticRegistry.global_config["default_time_window"]` 获取全局默认的时间窗口 ID（如 `TIME_LAST_30D`）。
 - **Stage 3：** 根据该 ID 从 `metadata_map` 中获取具体定义，构造时间范围填入 `time_range`。

- **Stage 3:** 追加提示 "未指定时间且无指标默认配置，已按系统全局设定展示最近 30 天数据"。

2. 强制过滤器注入 (Mandatory Filters Injection)

- **目的:**
自动应用业务定义的“有效性口径”（如：算销售额时必须排除取消订单），确保数据准确性。
- **执行过程:**
 - **遍历指标:** 遍历 `Skeleton Plan.metrics` 列表。
 - **获取规则:** 从 Registry 获取每个指标关联的 `default_filters`（逻辑过滤器 ID）。
 - **去重检查:** 针对每一个强制 Filter，检查 `Skeleton Plan.filters` 中是否已存在针对同一 `target_id` 的条件。
 - 若已存在（用户显式指定了），则**跳过**该强制 Filter（用户意图优先）。
 - 若不存在，则准备注入。
- **失败/降级策略:**
 - **策略: 自动修正 (Auto-Correction).** 属于 **隐性规则 (Implicit Rule)**，通常不视为 Warning。
 - **动作:**
 - **Stage 3:** 将强制 Filter 追加到 `Skeleton Plan.filters` 列表中。
 - **Stage 6:** 通常不提示用户，除非配置了“高敏感度”提示。

3. 维度补全 (Dimension Injection for Trend)

- **目的:**
确保趋势分析 (Trend Analysis) 意图下，Plan 中必然包含时间维度，否则无法绘制趋势图。
- **执行过程:**
 - **前置检查:** 若 `Skeleton Plan.intent` 不为 "TREND"，跳过此步。
 - **时间维度检查:** 遍历 `Skeleton Plan.dimensions`，检查是否存在 `is_time=True` 的维度。
 - **自动注入:** 若无时间维度，从当前主表 (Entity) 配置中读取 `default_time_field_id`。
- **失败/降级策略:**
 - **策略: 自动修正 (Auto-Correction).** 属于 **第一级: 警告 (Warning)**。
 - **动作:**
 - **Stage 3:** 将默认时间字段加入 `dimensions` 列表，设置默认 `time_grain`，并向 `Skeleton Plan.warnings` 追加提示（如 "已自动按 '订单日期' 进行趋势统计"）。
 - **Stage 6:** 告知用户使用了哪个时间字段进行趋势分析。

4. 排序与 Limit 补全 (Default Sort & Limit)

- 目的：
优化结果展示体验，并提供最后的系统级性能保护。
- 执行过程：
 - **Order By 补全**：若 `Skeleton Plan.order_by` 为空，根据 Intent（TREND 按时间升序，AGG 按指标降序）生成默认排序规则。
 - **Limit 补全**：若 `Skeleton Plan.limit` 为 `None`，读取全局配置 `pipeline_config.default_limit`。
- 失败/降级策略：
 - 策略：自动修正 (Auto-Correction)。属于 系统兜底 (System Fallback)。
 - 动作：
 - **Stage 3**：静默填入默认值。
 - **Stage 6**：对于 Limit 补全，若结果集确实被截断，前端需展示“仅展示前 100 条”的提示（通常由 Stage 5 的 `is_truncated` 标记控制，此处仅做参数补全）。

3.3.4 阶段总结

1. 流程概览

本阶段接收 Stage 2 的初步 Plan，依次执行结构完整性检查、术语存在性验证、权限复核（Security）、语义连通性校验（Metric/Dim 兼容性），并最终注入时间窗口与 Limit 等默认业务规则，输出一个逻辑自洽且安全的 Validated Plan。

2. 阶段流程逻辑伪代码：

代码块

```
1  def validate_and_normalize_plan(skeleton_plan, context, registry):
2      """
3      Stage 3 核心校验逻辑 (Final Version)
4      """
5      # -----
6      # 1. 权限复核 (Security First)
7      # -----
8      allowed_ids = registry.get_allowed_ids(context.role_id)
9      plan_ids = collect_all_ids(skeleton_plan)
10
11     if not plan_ids.issubset(allowed_ids):
12         raise PermissionDeniedError("Plan contains unauthorized IDs.")
13
14     # -----
15     # 2. 结构与存在性检查 (Sanity Check)
16     # -----
```

```

17     for term_id in plan_ids:
18         if not registry.exists(term_id):
19             raise ValidationError(f"Term not found: {term_id}")
20
21     if skeleton_plan.intent in ["AGG", "TREND"] and not skeleton_plan.metrics:
22         raise ValidationError("Aggregation query must have at least one
metric.")
23
24     # -----
25     # 3. 语义连通性 (Semantic Connectivity)
26     # -----
27     current_entity_id = None
28
29     # 3.1 主表推导与一致性检查
30     if skeleton_plan.metrics:
31         # A. 从指标推导主表
32         metric_entities = {registry.get_metric(m.id).entity_id for m in
skeleton_plan.metrics}
33         if len(metric_entities) > 1:
34             raise UnsupportedMultiFactError("MVP does not support multi-fact
queries.")
35         current_entity_id = list(metric_entities)[0]
36
37     elif skeleton_plan.dimensions:
38         # B. 从维度推导主表 (针对纯明细查询)
39         # 简单策略: 取第一个维度的 entity_id, 后续校验其他维度是否兼容
40         # 注意: 维度可能属于多个 Entity, 这里需 registry 支持 get_primary_entity
41         first_dim = registry.get_dimension(skeleton_plan.dimensions[0].id)
42         current_entity_id = first_dim.entity_id
43
44     if not current_entity_id:
45         raise ValidationError("Could not determine context entity from metrics
or dimensions.")
46
47     # 3.2 维度兼容性 (降级策略)
48     valid_dims = []
49     for dim in skeleton_plan.dimensions:
50         if registry.check_compatibility(current_entity_id, dim.id):
51             valid_dims.append(dim)
52         else:
53             skeleton_plan.warnings.append(f"Dimension '{dim.id}' is
incompatible with {current_entity_id}, removed.")
54     skeleton_plan.dimensions = valid_dims
55
56     # -----
57     # 4. 标准化注入 (Normalization)
58     # -----

```

```

59     # 4.1 时间窗口补全
60     if not skeleton_plan.time_range and skeleton_plan.metrics:
61         defaults = [registry.get_metric(m.id).default_time for m in
skeleton_plan.metrics]
62         first_window = defaults[0].time_window_id
63
64         if all(d.time_window_id == first_window for d in defaults):
65             skeleton_plan.time_range =
registry.resolve_time_window(first_window)
66             skeleton_plan.warnings.append(f"Time range not specified.
Defaulting to: {first_window}")
67         else:
68             raise AmbiguousTimeWindowError("Metrics have conflicting default
time windows.")
69
70     # 4.2 强制过滤器注入
71     if skeleton_plan.metrics:
72         for m in skeleton_plan.metrics:
73             mandatory_filters = registry.get_mandatory_filters(m.id)
74             for f in mandatory_filters:
75                 if not any(pf.id == f.id for pf in skeleton_plan.filters):
76                     skeleton_plan.filters.append(f)
77
78     # 4.3 默认排序与 Limit
79     if not skeleton_plan.order_by:
80         skeleton_plan.order_by = generate_default_sort(skeleton_plan)
81
82     if skeleton_plan.limit is None:
83         skeleton_plan.limit = pipeline_config.default_limit
84
85     return skeleton_plan

```

3. 最终输出示例 (Validated Plan)

代码块

```

1  {
2      "intent": "TREND",
3      "metrics": [
4          { "id": "METRIC_GMV", "compare_mode": "YOY" }
5      ],
6      "dimensions": [
7          { "id": "DIM_ORDER_DATE", "time_grain": "MONTH" },
8          { "id": "DIM_CITY" }
9      ],
10     "filters": [

```

```
11     { "id": "DIM_CITY", "op": "EQ", "values": ["Beijing"] }
12   ],
13   "time_range": {
14     "type": "ABSOLUTE",
15     "start": "2023-01-01",
16     "end": "2023-12-31",
17     "original_text": "Default Last 1 Year"
18   },
19   "order_by": [
20     { "id": "DIM_ORDER_DATE", "direction": "ASC" }
21   ],
22   "limit": 100,
23   "warnings": [
24     "Dimension 'DIM_PAYMENT' removed due to incompatibility."
25   ]
26 }
```

3.4 阶段四：SQL 生成阶段（Stage 4: PLAN → SQL）

3.4.1 阶段目标与职责

本阶段是纯粹的“编译器（Compiler）”工作。它不再猜测用户意图，而是将 Stage 3 输出的 `Validated Plan` 翻译为物理数据库可执行的 SQL 语句。

核心职责：

1. **语义翻译：**将 `METRIC_GMV` 翻译为 `SUM(t_orders.amount)`。
2. **关系连接：**根据 `semantic_core.yaml` 中的 `relationships` 配置生成 `JOIN` 子句。
3. **安全执法：**强制注入行级权限（RLS）和多租户隔离条件。

工具选型：

- **Pypika (Python Query Builder)：**专用于构建 SQL 字符串，轻量、无状态、防注入。

工具选型思路简述：

- **背景：**Pypika 由 Kayak（知名旅游搜索公司）开发，被 Apache Superset（最流行的开源 BI）作为核心 SQL 生成器使用。
- **核心优势：**
 - a. **纯 Python 对象操作：**`Query.from_('table').select('col')`，非常优雅。
 - b. **多方言支持：**原生支持 MySQL, PostgreSQL, Oracle, ClickHouse 等，切换数据库只需换一个配置。
 - c. **零依赖：**不需要连接数据库即可生成 SQL 字符串（完美契合 `POST /nl2sql/sql` 接

- d. **确定性**: 输入同样的 Plan，永远输出同样的 SQL。

备选方案对比（为什么不选别的）：

- **SQLAlchemy Core**: 过于厚重，更适合 ORM 场景（定义 Model 类）。我们需要动态生成查询，定义 Model 类太麻烦。
- **Python-SQL**: 语法稍显晦涩，社区活跃度不如 PyPika。
- **LangChain SQLChain**: 这是靠 LLM 写 SQL 的，不稳定，违反我们的设计原则。

3.4.2 输入与输出

输入 (Input)：

- `Validated Plan`：Stage 3 输出的标准化 Plan 对象。
- `SemanticRegistry`：提供物理表名、字段名、表达式和 Join 关系。
- `request_context`：提供 `user_id`，`role_id`，`tenant_id` 用于权限注入。

输出 (Output)：

- `final_sql`：一条完整的、可执行的 SQL 字符串 (String)。

3.4.3 处理流程 (Process Flow)

步骤一：语义映射与对象初始化 (Semantic Mapping)

子步骤 1.1：视图锚定与 Query 初始化 (View Resolution)

- **目的**：根据查询意图确定主语义视图 (FROM 子句)，并初始化 PyPika 构建器。
- **执行过程**:
 - a. **获取主实体**：读取 `plan.metrics[0].id`。若无指标（纯明细查询），读取 `plan.dimensions[0].id`。
 - b. **解析视图**:
 - 调用 `registry.get_metric_def(id)` 获取 `entity_id`。
 - 调用 `registry.get_entity_def(entity_id)` 读取配置项 `semantic_view`（例如 `v_sales_order_item`）。
 - c. **对象实例化**:
 - 创建 PyPika Table 对象：`base_table = Table('v_sales_order_item')`。
 - 初始化 Query 对象：`query = Query.from_(base_table)`。
 - 初始化安全集合：`involved_tables = {base_table}`（用于后续租户隔离注
- **失败/降级策略**：

- **策略：**熔断 (Fail Fast)，属于 **第三级：错误 (Config Mismatch)**。
- **动作：**若实体未配置 `semantic_view`，抛出 `PHYSICAL_MAPPING_ERROR`。

子步骤 1.2: 指标解析与投射 (Metrics Projection)

- **目的：**将 Plan 中的指标 ID 转换为 SELECT 列表中的聚合表达式。
- **执行过程：**
 - a. **遍历：**循环 `plan.metrics` 列表。
 - b. **读取配置：**从 Registry 获取指标的 `expression`（通常是 SQL 聚合公式，如 `SUM(amount)`）或 `field_name` + `agg_func`。
 - c. **构建表达式：**
 - **Case A (基础字段聚合)：**
 - `col = base_table[field_name]`
 - `expr = getattr(pypika.functions, agg_func)(col)`
 - **Case B (复杂表达式)：**
 - 若配置为 Raw SQL 表达式，使用 `pypika.CustomFunction` 或 `LiteralValue` 封装。
 - d. **别名绑定：**执行 `expr = expr.as_(metric.id)`，确保结果集列名与语义 ID 严格一致。
 - e. **注入 Query：**`query = query.select(expr)`。
- **失败/降级策略：**
 - **策略：**熔断 (Fail Fast)。
 - **动作：**若聚合函数不支持，抛出 `UNSUPPORTED_AGGREGATION`。

子步骤 1.3: 维度解析与时间归一化 (Dimensions & Time Normalization)

- **目的：**处理 SELECT 和 GROUP BY，并通过适配器解决不同数据库的时间截断差异。
- **执行过程：**
 - a. **遍历：**循环 `plan.dimensions` 列表。
 - b. **字段映射：**
 - 从 Registry 获取维度对应的 `field_name`。
 - 构建字段对象：`col = base_table[field_name]`。
 - c. **时间粒度适配 (Dialect Delegation)：**
 - 检查 `dim.time_grain` 是否非空（如 "MONTH"）。
 - **若非空：**
 1. 实例化：`adapter = DialectAdapter(pipeline_config.db_type)`

2. 调用: `adapter.get_time_truncation_sql(...)`

- **技术实现规范 (Implementation Spec):**

该方法必须基于以下静态映射表生成 SQL 片段（严禁 LLM 临场发挥）：

代码块

```
1  # core/dialect_adapter.py
2  TIME_TRUNCATION_MAP = {
3      "mysql": {
4          "DAY": "DATE_FORMAT({col}, '%Y-%m-%d')",
5          "WEEK": "DATE_FORMAT({col}, '%Y-%u')",
6          "MONTH": "DATE_FORMAT({col}, '%Y-%m-01')",
7          "QUARTER": "MAKEDATE(YEAR({col}), 1) + INTERVAL
8          QUARTER({col})-1 QUARTER",
9          "YEAR": "DATE_FORMAT({col}, '%Y-01-01')",
10     },
11     "postgresql": {
12         "DAY": "DATE_TRUNC('day', {col})",
13         "WEEK": "DATE_TRUNC('week', {col})",
14         "MONTH": "DATE_TRUNC('month', {col})",
15         "QUARTER": "DATE_TRUNC('quarter', {col})",
16         "YEAR": "DATE_TRUNC('year', {col})"
17     }
18 }
```

⚠ 类型强约束 (Critical Data Type Constraint):

- **前提条件:** 上述 Map 中的截断函数（如 `DATE_TRUNC` , `DATE_FORMAT`）仅适用于数据库原生的 `DATE` / `DATETIME` / `TIMESTAMP` 类型。
- **禁止事项:** 严禁对 `VARCHAR` / `STRING` 类型的日期字段直接应用此逻辑。若底层表设计不规范（用字符串存日期），必须在 ETL 层治理，严禁在 SQL 生成层做 `STR_TO_DATE` 等隐式转换，以防止索引失效和性能崩塌。

- 将返回的片段封装为 PyPika 表达式 `col_expr`。

- 若为空: `col_expr = col`。

d. 注入 Query:

- `query = query.select(col_expr.as_(dim.id))`
- `query = query.groupby(col_expr)`

- **失败/降级策略:**

- **策略:** 熔断 (Fail Fast)。

- **动作:** 若 `pipeline_config.db_type` 不在 MAP 支持列表中, 或粒度不支持, 抛出 `UNSUPPORTED_DIALECT_GRAIN`。

步骤二: 视图归属校验 (View Scope Validation)

- **目的:** 确保 PLAN 中请求的所有指标和维度, 都属于同一个物理语义视图 (宽表), 防止跨视图查询。
- **【架构设计说明 (Architecture Note)】:**
 - 本项目采用 **OBT (One Big Table) / 语义视图** 策略。
 - **零 Join 假设:** 系统假定所有需要的维度字段 (如 `city_name`, `product_name`) 都已经通过 ETL 预处理宽表化, 直接存在于 `semantic_view` 中。
 - **因此:** 本阶段 **不需要** 执行复杂的图遍历 (Graph Traversal) 或多跳 Join 路径推导 (Snowflake Schema Resolution)。只需校验字段是否“同属一表”即可。
- **执行过程:**
 - a. **获取主视图:** 根据 `plan.metrics[0]` 对应的 Entity, 确定主语义视图 (如 `v_sales_order_item`)。
 - b. **维度校验:** 遍历 `plan.dimensions` 和 `plan.filters`, 检查每个 Dimension 对应的 Entity 是否与主视图一致 (或在 YAML 配置中明确归属于该 Entity)。
 - c. **判定:**
 - 若所有字段均归属同一视图 -> **校验通过**。
 - 若出现跨视图字段 (如同时查“订单行”和“员工档案”) -> **触发熔断**。
- **失败/降级策略:**
 - **策略:** 熔断 (Fail Fast), 属于 **第三级: 错误 (Config Missing)**。
 - **动作:** 抛出 `UNSUPPORTED_CROSS_VIEW_QUERY` 异常。MVP 阶段不支持跨宽表 Join, 需提示用户拆分问题

步骤三: 过滤条件构建 (Filter Construction)

子步骤 3.1: 时间范围转换 (Time Range)

- **目的:** 将标准时间窗口转换为 SQL `WHERE` 条件。
- **执行过程:**
 - a. **检查:** 若 `plan.time_range` 为 None, 跳过。
 - b. **获取字段:** 确定主表的时间字段 (如 `order_date`)。
 - c. **生成条件:**
 - Start: `base_table[time_col] >= plan.time_range.start`

- End: `base_table[time_col] <= plan.time_range.end`

d. 存入：将这两个 Criterion 对象存入 `where_criteria` 列表。

- 失败/降级策略：

- 策略：忽略 (Ignore)，属于 **第一级：警告**。
- 动作：若时间格式解析失败，记录 Warning，不生成时间过滤（退化为全量查询，但通常 Stage 3 会拦住）。

子步骤 3.2：普通过滤器转换 (Standard Filters)

- 目的：将 JSON Filter 转换为 PyPika Criterion。

- 执行过程：

a. 遍历：循环 `plan.filters`。

b. 区分类型：

- 若 `id` 是 Dimension -> 目标列表为 `where_criteria`。
- 若 `id` 是 Metric -> 目标列表为 `having_criteria`。

c. 操作符映射：

- EQ -> `col == val`
- IN -> `col.isin(vals)`
- LIKE -> `col.like(val)` (注意处理通配符)
- ... (其他操作符映射)

d. 存入：将生成的 Criterion 加入对应列表。

- 失败/降级策略：

- 策略：熔断，属于 **第三级：错误**。
- 动作：若遇到不支持的操作符，抛出 `UNSUPPORTED_OPERATOR`。

步骤四：安全注入 (Security Injection)

子步骤 4.1：全链路租户隔离 (Universal Tenant Isolation)

- 目的：防止多租户数据泄露，覆盖主表及所有 Join 的维表。

- 执行过程：

a. 遍历：循环 `involved_tables` 集合（包含 Base Table 和所有 Joined Tables）。

b. 检查结构：调用 `registry.get_table_schema(table_name)` 检查物理列中是否包含 `tenant_id`。

c. 强制注入：

- 若存在 `tenant_id`：

- 生成条件: `table.tenant_id == request_context.tenant_id`。
- 加入 `where_criteria`。
- 失败/降级策略:
 - 策略: 强制执行, 无降级。
 - 动作: 这是安全底线, 不依赖任何外部配置, 纯物理结构检查。

子步骤 4.2: 行级权限注入 (RLS Injection)

- 目的: 将抽象的角色权限转换为物理 SQL 过滤。
- 执行过程:
 - a. 获取策略: 根据 `role_id` 读取 `semantic_security.yaml` 中的 SQL 片段模板 (如 `region_id in ({{ valid_regions }})`)。
 - b. 模板渲染: 使用 `request_context` 里的变量 (如 `user_id`) 渲染模板。
 - c. 解析为对象:
 - 注意: 这里 PyPika 很难直接解析复杂的 Raw SQL 字符串。
 - MVP 方案: 使用 `pypika.CustomFilter` 或直接在 `Where` 中注入 `Criterion.all([..., output_sql])`。
 - d. 合并: 将生成的 RLS Criterion 加入 `where_criteria`。
- 失败/降级策略:
 - 策略: 熔断 (Fail Fast), 属于 **第三级: 错误 (Security Risk)**。
 - 动作: 若 RLS 模板渲染失败, **必须抛出异常**, 严禁生成无 RLS 的 SQL。

步骤五: 最终组装 (Final Assembly)

子步骤 5.1: 组装与方言渲染

- 目的: 生成最终字符串。
- 执行过程:
 - a. 应用 WHERE:
 - 若 `where_criteria` 非空: `query = query.where(Criterion.all(where_criteria))`。
 - b. 应用 HAVING:
 - 若 `having_criteria` 非空: `query = query.having(Criterion.all(having_criteria))`。
 - c. 应用 ORDER BY:
 - 根据 `plan.order_by` 追加排序。

d. 应用 LIMIT:

- `query = query.limit(plan.limit)`。

e. 渲染字符串:

- `final_sql = query.get_sql(quote_char='\"') (默认为 ANSI SQL)。`
- *Hook*: 若是 ClickHouse, 可能需移除引号或调整格式。

• 失败/降级策略:

- **策略**: 返回空结果, 属于 **第一级: 警告**。
- **动作**: 若生成的 SQL 为空字符串 (理论不应发生), Stage 5 将无法执行。

3.4.4 阶段总结

本阶段的数据流转是一个严密的 “**翻译与组装**” 过程, 从上到下依次经过以下环节:

1. 流程概览 (Process Review)

本阶段接收 Validated Plan, 利用 PyPika 进行确定性的语义翻译 (映射/连接/过滤), 并在最终渲染前强制注入多租户与 RLS 安全围栏, 输出绝对安全的可执行 SQL。

2. 核心逻辑伪代码 (Core Logic Pseudo-Code)

代码块

```
1  def generate_sql(validated_plan, context, registry):
2      # --- 1. 语义对象映射 (Semantic Mapping) ---
3      # 确定主表 (FROM)
4      base_table_def = registry.get_table_def(validated_plan.metrics[0])
5      query = Query.from_(Table(base_table_def.physical_name))
6
7      # 映射指标 (SELECT)
8      for metric in validated_plan.metrics:
9          expr = getattr(pypika.functions, metric.agg_func)
10         (Field(metric.field_name))
11         query = query.select(expr.as_(metric.id))
12
13     # 映射维度 (SELECT + GROUP BY + TimeGrain)
14     for dim in validated_plan.dimensions:
15         col = Field(dim.field_name)
16         # 方言处理: 时间截断
17         if dim.time_grain:
18             col = dialect_adapter.truncate_time(col, dim.time_grain,
19 pipeline_config.db_type)
20         query = query.select(col).groupby(col)
21
22     # --- 2. 拓扑连接 (Join Resolution) ---
23     # 计算需要 Join 的表
```

```

22     join_targets = calculate_join_targets(validated_plan, base_table_def)
23     for target in join_targets:
24         # 查找配置的 Join 路径
25         relation = registry.get_relation(base_entity, target.entity)
26         query = query.join(Table(target.physical_name),
how=JoinType.left).on(relation.condition_expr)
27
28     # --- 3. 过滤构建 (Filters) ---
29     where_conditions = []
30     # 时间范围
31     if validated_plan.time_range:
32         where_conditions.extend(build_time_criteria(validated_plan.time_range))
33     # 普通 Filter
34     for f in validated_plan.filters:
35         where_conditions.append(build_criteria(f))
36
37     # --- 4. 安全注入 (Security Injection - Critical) ---
38     # 4.1 全链路租户隔离
39     all_involved_tables = [base_table_def] + join_targets
40     for t in all_involved_tables:
41         if t.has_column("tenant_id"):
42             where_conditions.append(Field("tenant_id") == context.tenant_id)
43
44     # 4.2 RLS 行级权限
45     rls_sql_fragment = registry.get_qls_policy(context.role_id)
46     if rls_sql_fragment:
47         where_conditions.append(CustomCriterion(rls_sql_fragment))
48
49     # --- 5. 组装与渲染 ---
50     query = query.where(Criterion.all(where_conditions))
51     query = query.limit(validated_plan.limit)
52
53     return query.get_sql()

```

3. 输出产物 (安全可执行的SQL 字符串)

最终输出示例：（假设查询：“华北区”最近30天的各城市销售额，且当前用户只能看“自营”渠道的数据。）

代码块

```

1  SELECT
2      DATE_FORMAT(t1.order_date, '%Y-%m-01') AS "DIM_DATE_MONTH", -- 时间粒度处理
3      t2.city_name AS "DIM_CITY", -- 维度映射
4      SUM(t1.total_amount) AS "METRIC_GMV" -- 指标聚合
5  FROM
6      fact_sales_orders t1 -- 主表确定

```

```

7      LEFT JOIN dim_geo t2 ON t1.geo_id = t2.id          -- 关系连接生成
8  WHERE
9      t1.order_date >= '2023-10-01'                    -- 时间范围转换
10     AND t1.order_date <= '2023-10-30'
11     AND t2.region_name = 'North China'                -- 普通维度过滤
12
13     -- 【安全注入层】
14     AND t1.tenant_id = 'tenant_001'                    -- 租户隔离（主
    表）
15     AND t2.tenant_id = 'tenant_001'                    -- 租户隔离（维
    表）
16     AND t1.channel_type = 'SELF_OPERATED'             -- RLS 行级权限
    注入
17 GROUP BY
18     DATE_FORMAT(t1.order_date, '%Y-%m-01'),
19     t2.city_name
20 LIMIT 100                                              -- 默认 Limit
    保护

```

3.5 阶段五：SQL 执行阶段（Stage 5: SQL → Result）

3.5.1 阶段目标与职责

Stage 5 是 NL2SQL 链路中的“**执行沙箱（Execution Sandbox）**”。它完全不关心 SQL 的业务含义，只负责在一个**受限、安全、可观测**的环境中运行 SQL，并将物理结果转化为前端友好的 JSON 格式。

核心原则：

- **资源闭环 (Resource Safety)**：确保数据库连接在任何情况下（包括崩溃、超时）都能 100% 自动归还，杜绝连接池耗尽。
- **零信任防御 (Zero Trust)**：默认不信任输入的 SQL 是高效的，强制施加会话级超时和只读限制。
- **防崩溃 (Crash Prevention)**：通过硬截断（Hard Limit）防止大结果集导致后端内存溢出 (OOM)。

3.5.2 输入与输出

输入 (Input)：

- `final_sql` (str)：Stage 4 生成的、针对特定方言的 SQL 字符串。

- `user_context`：提供 `tenant_id`（用于多租户连接路由）和 `request_id`（用于日志追踪）。
- `pipeline_config`：全局配置，提供 `max_result_rows`（最大行数）和 `execution_timeout`（超时时间）。
- `.env` (Environment)：提供数据库连接所需的敏感信息（Host, Port, User, Password），由 `db_connector` 组件内部读取。

输出 (Output)：

- `ExecutionResult` 对象：包含标准化后的数据（columns, rows）、状态（status）和元数据（execution_meta）。

3.5.3 处理流程 (Process Flow)

本阶段采用“上下文管理器 (Context Manager)”模式，确保资源生命周期严格受控。

步骤一：资源准备与方言适配 (Resource & Adapter Setup)

子步骤 1.1：实例化适配器(Adapter Initialization)

- **目的：**根据全局配置确定当前数据库的方言策略（MySQL vs PG），屏蔽语法差异。
- **执行过程**
 - a. 读取 `pipeline_config.db_type`（字符串，如 "mysql", "postgresql"）。
 - b. 实例化 `core.dialect_adapter.DialectAdapter` 对象。
 - c. 验证该方言类型是否被 Adapter 支持。
- **失败/降级策略：**
 - **策略：**熔断 (Fail Fast)，属于 **第三级：错误 (Config Error)**。
 - **动作：**若 `db_type` 不在支持列表中，抛出 `UNSUPPORTED_DB_TYPE` 异常，终止流程。

子步骤 1.2：引擎获取 (Engine Retrieval)

- **目的：**从连接池管理器中获取针对当前租户的 SQLAlchemy Engine 对象。
- **执行过程：**
 - a. 调用 `core.db_connector.get_engine(user_context.tenant_id)`。
 - b. `db_connector` 内部检查是否已初始化：
 - 若未初始化，读取 `.env` 中的 `DB_HOST`，`DB_USER`，`DB_PASSWORD` 等，创建全局唯一的 `Engine` 单例（带 QueuePool）。
 - 若已初始化，直接返回单例引用。
- **失败/降级策略：**
 - **策略：**熔断 (Fail Fast)，属于 **第三级：错误 (Config Error)**。

- **动作：**若 `.env` 缺失关键配置，抛出 `DB_CONFIGURATION_MISSING`，终止流程。

步骤二：安全执行沙箱 (Secure Execution Sandbox) —— 核心步骤

子步骤 2.1：安全上下文开启 (Context Guard)

- **目的：**建立物理连接，并确保连接在任何情况下都能自动归还连接池。
- **执行过程：**
 - a. 记录 `start_time = time.time()`。
 - b. **关键动作：**执行 `with engine.connect() as conn:`。
 - c. 代码进入 `with` 块内部，获取到活跃连接 `conn`。
- **失败/降级策略：**
 - **策略：**异常映射，属于 **第三级：错误 (Runtime Error)**。
 - **动作：**若连接池耗尽或数据库宕机，SQLAlchemy 会抛出 `OperationalError`。Stage 5 捕获此异常，映射为 `DB_CONNECTION_ERROR`，提示“服务繁忙或数据库不可用”。

子步骤 2.2：会话防御注入 (Session Guardrails)

- **目的：**在执行用户查询前，先执行“环境安保”指令（超时/只读）。
- **执行过程：**
 - a. 调用 `adapter.get_session_setup_sql(timeout_ms=config.timeout, read_only=True)`。
 - b. 获取指令列表（例如 MySQL 返回 `["SET SESSION MAX_EXECUTION_TIME=50000"]`）。
 - c. 遍历列表，依次执行 `conn.execute(text(cmd))`。
- **失败/降级策略：**
 - **策略：**熔断，属于 **第三级：错误**。
 - **动作：**若防御指令执行失败（如权限不足），抛出 `DB_PERMISSION_DENIED`，禁止后续查询。

子步骤 2.3：查询执行与硬截断 (Execution & Truncation)

- **目的：**发送 SQL 并安全获取有限的结果集。
- **执行过程：**
 - a. **发送查询：**执行 `cursor = conn.execute(text(final_sql))`。
 - b. **获取元数据：**调用 `cursor.keys()` 获取列名列表 `columns`。
 - c. **防御性拉取：**
 - 读取限制：`limit = pipeline_config.max_result_rows`。
 - 执行拉取：`raw_rows = cursor.fetchmany(limit + 1)`（多取1条用于探测）。

d. 截断判定:

- 若 `len(raw_rows) > limit`: 将 `raw_rows` 的最后一行移除, 并设置局部变量 `is_truncated = True`。
- 否则: 设置 `is_truncated = False`。

• 失败/降级策略:

- 策略: 异常映射, 属于 **第二级: 澄清** / **第三级: 错误**。
- 动作:
 - `TimeoutError` -> 抛出 `SQL_EXECUTION_TIMEOUT`。
 - `ProgrammingError` (如表不存在) -> 抛出 `INTERNAL_SCHEMA_MISMATCH` (不暴露原始堆栈)。

步骤三: 数据清洗与序列化 (Data Sanitization)

子步骤 3.1: 类型标准化 (Type Normalization)

- 目的: 将数据库物理类型转换为 JSON 兼容的基础类型。

• 执行过程:

- a. 遍历 `raw_rows` 的每一行。
- b. 对每个字段值进行 `isinstance` 判断:
 - `Decimal`: 转 `float(val)` (保留2位小数)。
 - `datetime/date`: 调用 `val.isoformat()` 转字符串。
 - `UUID`: 调用 `str(val)`。
 - `bytes`: 忽略或转 Base64 字符串 (MVP 默认转 `"<BINARY>"` 占位符)。
 - `None`: 保持 `None`。
- c. 生成清洗后的 `clean_rows`。

• 失败/降级策略:

- 策略: 自动修正 (Auto-Correction), 属于 **第一级: 警告**。
- 动作: 若遇到无法识别的自定义类型, 强制转换为 `str(val)`, 确保接口不报错。

步骤四: 结果封装与审计 (Result Encapsulation)

子步骤 4.1: 审计日志与返回 (Logging & Return)

- 目的: 记录性能指标并输出最终对象。

• 执行过程:

- a. 计算耗时: `latency = (time.time() - start_time) * 1000`。

b. 慢查询记录：若 `latency > 2000` (2秒)，调用 `log_manager.warning` 记录慢 SQL（注意：仅在日志中保留 SQL，不返回给前端）。

c. 对象构造：

- 实例化 `schemas.result.ExecutionResult`。
- 填入 `status="SUCCESS"`，`data={rows, columns, is_truncated}`，`meta={latency, row_count}`。

d. 返回该对象。

• 失败/降级策略：

- 无（纯内存操作）。

3.5.5 阶段总结

1. 流程概览 (Process Review)

本阶段利用 `DialectAdapter` 屏蔽数据库差异，通过 `with` 语句构建绝对安全的执行上下文（防连接泄漏），依次实施会话级防御（超时/只读）与结果集硬截断，最终将物理数据清洗为标准化的 `ExecutionResult` 对象。

2. 核心逻辑伪代码 (Core Logic Pseudo-Code)

代码块

```
1  def execute_sql_and_fetch_result(final_sql, user_context):# --- 1. 资源准备与适
    配 (Resource Setup) ---
2      adapter = DialectAdapter(pipeline_config.db_type)
3      # 获取单例引擎（内部已加载 .env 配置）
4      db_engine = db_connector.get_engine(user_context.tenant_id)
5
6      start_time = time.time()
7
8      # --- 2. 安全沙箱执行 (Secure Sandbox) ---# 强制使用 Context Manager，确保连接
    100% 归还连接池try:
9          with db_engine.connect() as conn:
10
11              # --- 3. 会话级防御 (Session Guardrails) ---# 根据方言注入防御指令（如
    MySQL SET MAX_EXECUTION_TIME）
12              setup_cmds = adapter.get_session_setup_sql(
13                  timeout_ms=pipeline_config.execution_timeout,
14                  read_only=True
15              )
16              for cmd in setup_cmds:
17                  conn.execute(text(cmd))
18
```

```

19         # --- 4. 执行与硬截断 (Execution & Hard Limit) ---
20         cursor = conn.execute(text(final_sql))
21
22         # 防 OOM: 多取一条用于探测是否溢出
23         limit = pipeline_config.max_result_rows
24         raw_rows = cursor.fetchmany(limit + 1)
25
26         is_truncated = False if len(raw_rows) > limit:
27             raw_rows.pop() # 移除探测用的最后一条
28             is_truncated = True
29
30         columns = cursor.keys()
31
32     except Exception as e:
33         # 错误映射: 隐藏原始堆栈, 转为业务错误码
34         raise map_db_error(e)
35
36     # --- 5. 清洗与封装 (Sanitization) --- # 类型标准化: Decimal->float, Date->str
37     clean_rows = sanitize_rows(raw_rows)
38
39     return ExecutionResult(
40         status="SUCCESS",
41         data={
42             "columns": [{"name": c} for c in columns],
43             "rows": clean_rows,
44             "is_truncated": is_truncated
45         },
46         execution_meta={
47             "latency_ms": (time.time() - start_time) * 1000,
48             "row_count": len(clean_rows)
49         }
50     )

```

3. 最终输出示例 (Final Output Example)

(假设场景: 执行了查询“近3个月每月的GMV趋势”, 数据库返回了 `Decimal` 和 `Date` 类型, 且未触发截断。)

代码块

```

1  {
2    "status": "SUCCESS",
3    "data": {
4      "columns": [
5        {"name": "DIM_DATE_MONTH", "type": "STRING"},
6        {"name": "METRIC_GMV", "type": "FLOAT"}
7      ],
8      "rows": [

```

```
9      // 类型清洗: Date 对象已转为 ISO 字符串, Decimal 已转为 float
10      ["2023-10-01", 15000.50],
11      ["2023-11-01", 18200.00],
12      ["2023-12-01", 21000.00]
13  ],
14      "is_truncated": false // 未超过 max_result_rows 限制
15  },
16      "execution_meta": {
17          "latency_ms": 145, // 执行耗时 "row_count": 3,
18          "executed_at": "2023-12-05T10:00:00Z"
19      }
20  }
```

3.6 阶段六：最终响应生成（Stage 6: Result → Answer）

3.6.1 阶段目标与职责

本阶段是流水线的“总出口”。由于上游可能并发执行了多个子查询，本阶段的核心职责从“解释一个结果”升级为“综合汇报”。

核心职责：

- **多源汇总 (Multi-Result Aggregation):** 接收 Orchestrator 汇总的多个执行结果（成功的数据或失败的异常）。
- **部分成功处理 (Partial Success Handling):** 当“查销量成功”但“查库存失败”时，不能直接报错，而应**“展示成功的数据，并解释失败的部分”**。
- **综合洞察 (Synthesized Insight):** 将多个子查询的数据表格拼接，让 LLM 生成一份综合性的分析总结。

3.6.2 输入与输出 (Stage I/O)

输入 (Input) - [变更]:

- `batch_results` (List): 由 Orchestrator 聚合的列表，每个元素包含：
 - `sub_query`: 原始子查询描述（用于生成标题）。
 - `status`: "SUCCESS" | "ERROR"。
 - `payload`: `ExecutionResult` (成功时) 或 `PipelineError` (失败时)。
- `user_query`: 原始用户问题。

输出 (Output) - [变更]:

- `FinalAnswer` 对象:
 - `answer_text` (str): 综合分析文案。
 - `data_list` (List[Dict]): **[结构变更]** 结构化数据列表, 供前端渲染多个图表。
 - 包含: `{ "sub_query_id": "...", "title": "...", "data": ResultData, "error": null }`
 - `status` (str): `SUCCESS` (全成) | `PARTIAL_SUCCESS` (部分成) | `ALL_FAILED` (全败)。

3.6.3 处理流程 (Process Flow) —— 终极合并版

本阶段的核心是“多路聚合”与“动态响应”。代码需处理从“全胜”到“全败”的各种中间状态。

步骤一：结果分拣与路由 (Result Sorting & Routing)

- 目的: 遍历 Orchestrator 返回的 `batch_results`, 统计成功/失败数量, 决定响应策略。
- 执行过程:
 - a. 初始化 `success_items = []`, `failed_items = []`。
 - b. 遍历列表:
 - 若 `status == SUCCESS`: 加入 `success_items`, 提取 `ExecutionResult`。
 - 若 `status == ERROR`: 加入 `failed_items`, 提取 `PipelineError`。
 - c. 模式判定:
 - **Case A (全败):** `len(success_items) == 0` -> 跳转至 步骤三: 异常响应生成。
 - **Case B (全胜/部分胜):** `len(success_items) > 0` -> 跳转至 步骤二: 综合洞察生成。

步骤二：综合洞察生成 (Synthesized Insight) —— 成功/部分成功路径

- 目的: 将多个数据表格拼接给 LLM, 生成一份回答; 同时处理部分失败的告警。
- 执行过程:
 - a. 多表 Markdown 构建 (Multi-table Construction):
 - 初始化 `multi_table_markdown = ""`。
 - 遍历 `success_items` (index `i`, result `res`):
 - 读取限制: `limit = pipeline_config.max_llm_rows`。
 - 数据截断:
 - `display_rows = res.rows[:limit]`

- 若 `len(res.rows) > limit`，生成尾注：`footer = f"\n...(共 {total} 条, 仅展示前 {limit} 条)"`。
- **格式转换：**将 `display_rows` 和 `columns` 转换为 Markdown Table。
- **拼接：**追加标题与表格：
- `codeText`

代码块

```
1   ### 子查询 {i+1}: {sub_query.description}
2   {markdown_table}
3   {footer}
```

b. 错误信息注入 (Partial Failure Injection):

- 若 `failed_items` 非空，在 Markdown 末尾追加警示：
`### ⚠️ 部分数据缺失：子查询 "{failed_sub_query}" 执行失败，原因：{error.message}。`

c. Prompt 构建与生成:

- 加载模板：`PROMPT_DATA_INSIGHT` (见附录 A)。
- 变量注入：传入 `user_query`，`plan_summary`，`multi_table_markdown`。
- 调用 LLM：`await llm_client.chat(...)` 获取 `answer_text`。

d. 对象组装:

- 构造 `FinalAnswer`。
- `status`: 若 `failed_items` 为空则 `SUCCESS`，否则 `PARTIAL_SUCCESS`。
- `data_list`: 将所有 `success_items` 封装为前端所需的结构化列表。

步骤三：异常响应生成 (Error Handling) —— 全败路径

- **目的：**当所有子查询都失败时，选择一个“最主要”的错误，生成友好的报错或追问。
- **执行过程：**

a. 主因筛选 (Primary Error Selection):

- 从 `failed_items` 中选择优先级最高的错误（优先级：`PERMISSION_DENIED` > `AMBIGUOUS` > `TIMEOUT` > `SYSTEM_ERROR`）。
- 令其为 `primary_error`。

b. 异常分类映射 (Error Classification) —— [保留原版逻辑]:

- 读取 `primary_error.code`。

- **Type A (需澄清):** `AMBIGUOUS_TIME`, `MISSING_METRIC`, `AMBIGUOUS_ENTITY`
 - -> 进入子步骤 3.1: 智能追问生成。
- **Type B (直接拒绝):** `PERMISSION_DENIED`, `UNSUPPORTED_INTENT`, `SENSITIVE_DATA`
 - -> 进入子步骤 3.2: 静态模板响应。
- **Type C (系统故障):** `DB_CONNECTION_ERROR`, `TIMEOUT`, `UNKNOWN`
 - -> 进入子步骤 3.2: 静态模板响应。
- **子步骤 3.1: 智能追问生成 (Clarification Generation)**
 - **动作:**
 - 加载模板 `PROMPT_CLARIFICATION`。
 - 注入 `user_query` 和 `primary_error.message`。
 - 调用 LLM 生成追问文案。
 - 设置 `status = "CLARIFICATION_NEEDED"`。
 - **降级:** 若 LLM 调用失败, 直接返回 `primary_error.message`。
- **子步骤 3.2: 静态模板响应 (Static Template Response)**
 - **动作:**
 - 若为 **Type B**: 读取模板 `TEMPLATE_REFUSAL`。
 - 若为 **Type C**: 读取模板 `TEMPLATE_SYSTEM_ERROR`。
 - 格式化生成文案。
 - 设置 `status = "ERROR"`。

步骤四: 最终封装 (Final Encapsulation)

- **目的:** 统一 API 输出契约。
- **执行过程:**
 - a. 实例化 `FinalAnswer` 对象。
 - b. 字段填充:
 - `status`: 来自 步骤二 或 步骤三 的计算结果。
 - `answer_text`: LLM 生成的分析、追问或错误提示。
 - `data_list`: 成功的数据列表 (全败时为空列表)。
 - `error`: 若全败, 填入 `primary_error.code` 供前端埋点。
 - c. 返回对象。

整体流程示意图：



4. 异常处理与流转机制（Exception Handling & Flow Control）

4.1.1 设计思想

本系统遵循 "Fail Fast, Degrade Gracefully"（快速失败，优雅降级）与 "Determinism First"（确定性优先）的原则。我们将运行时的异常情况严格划分为三个等级，不同等级对应不同的流转路径和用户反馈策略。

1. 快速失败 (Fail Fast):

- 在 Stage 2/3 尽早发现问题（如幻觉 ID、逻辑冲突），避免将错误传导至 SQL 执行层消耗资源。

2. 零猜测与零并发 (No Guessing & No Branching):

- 定义：**当用户意图存在歧义（Ambiguity）且无法通过默认规则消除时（例如匹配到多个指标且无优先级）。
- 策略：**
 - 严禁瞎猜：**后端不得根据“热度”或“随机”选择一个指标继续执行。
 - 严禁并发：**后端不得同时生成多条 SQL 并行查询所有可能性。
 - 强制熔断：**必须在 Stage 3 立即终止流程，**跳过 Stage 4/5**，直接进入 Stage 6 触发澄清机制。

3. 优雅降级 (Graceful Degradation):

- 对于非致命的缺失（如未指定时间），允许使用默认值补全，但必须通过 Warning 告知用户，保证系统的透明度。

4.1.2 三级响应机制 (The Three-Level Response)

第一级：警告 (Warning) —— "能跑，但有瑕疵"

- 定义：**用户的查询意图不完整或包含轻微错误，但系统可以通过默认值或忽略部分条件来完成查询。
- 典型场景：**
 - 用户未指定时间范围（系统自动补全默认 30 天）。
 - 用户提及了不存在的指标（系统自动忽略该指标，查询其余部分）。
- 流转路径：**
 - Stage 2/3：**发现问题 -> 修正 Plan -> 将修正动作写入 `Plan.warnings` -> **继续流转**。
 - Stage 4/5：**正常生成并执行 SQL。
 - Stage 6：**生成回答时，在末尾附带提示（如：“注：未指定时间，已默认展示最近 30 天数数据”）。
- HTTP 状态：** `200 OK`

第二级：澄清 (Clarification) —— "跑不下去，需要追问"

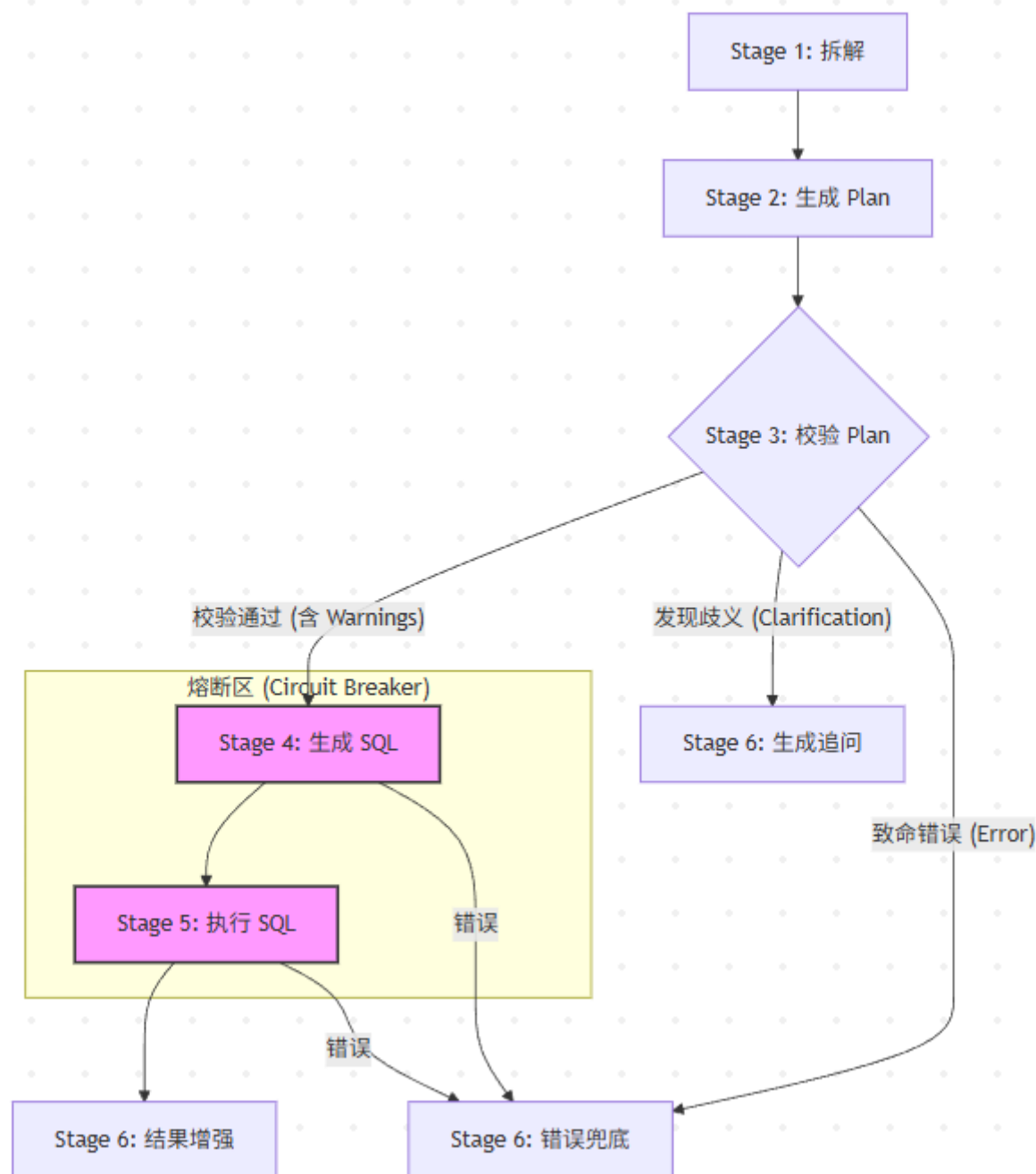
- **定义：**用户的查询存在无法自动消除的歧义，或者缺少生成 SQL 必须的核心条件（如 `metrics` 为空）。
- **典型场景：**
 - 关键词匹配到多个指标且无默认优先级（如“业绩”对应“个人业绩”和“部门业绩”）。
 - 意图为“对比”但未提供对比对象。
- **流转路径 (熔断机制)：**
 - **Stage 3：**校验失败 -> 抛出业务异常（如 `AmbiguousIntentError`），携带候选集 -> **中断流程**（跳过 Stage 4/5）。
 - **Stage 6：**捕获异常 -> 读取候选集 -> 生成自然语言追问（如：“请问您是指...还是...？”）。
- **HTTP 状态：** `200 OK`（业务状态码标记为 `NEED_CLARIFICATION`）

第三级：错误 (Error) —— "系统故障或非法请求"

- **定义：**发生了技术性故障，或者用户请求完全超出系统能力边界。
- **典型场景：**
 - **非法请求：**用户问“今天天气如何”（非数据查询）。
 - **权限阻断：**用户试图访问无权数据（`PermissionDenied`）。
 - **系统故障：**数据库连接超时、SQL 执行报错、LLM 服务不可用。
- **流转路径：**
 - **任意 Stage：**抛出致命异常 -> **中断流程**。
 - **Stage 6 (Global Error Handler)：**捕获异常 -> 映射为友好的错误提示（隐藏堆栈信息）。
- **HTTP 状态：** `400 Bad Request` 或 `500 Internal Server Error`

4.1.3 异常流转视图 (Exception Flow Diagram)

为了确保开发人员理解“跳过 SQL 执行”的逻辑，特定义以下流转规则：



5. 服务接口与可观测性 (Service Interface & Observability)

本章定义了 NL2SQL 服务对外的 API 契约与调试机制。为了支持测试驱动开发 (TDD) 和分层验证，系统摒弃了单一黑盒接口的设计模式，转而暴露流水线的关键中间状态。

5.1 接口设计策略 (Design Strategy)

为了降低调试成本并加速问题定位，系统采用**分层接口暴露策略**：

1. **解耦验证**: 允许独立验证“语义理解 (LLM)”与“代码生成 (Logic)”两个环节，避免因单一全链路接口导致的错误归因困难（例如：无法区分是 LLM 理解错了意图，还是 SQL 拼写错误）。
2. **成本控制**: 在进行 SQL 生成器或执行层的回归测试时，允许跳过昂贵的 LLM 调用阶段 (Mock Plan)，显著降低 Token 消耗与时间成本。

5.2 核心端点定义 (Core Endpoints)

在 `main.py` (FastAPI) 中，必须注册以下三个核心路由，分别对应流水线的不同切面。

5.2.1 语义理解接口 (The "Brain" Check)

- **Path**: `POST /nl2sql/plan`
- **覆盖范围**: Stage 1 (拆解) + Stage 2 (解析) + Stage 3 (校验)
- **输入**: 自然语言问题 (String) + 用户上下文 (Context)
- **输出**: `Validated Plan` (JSON)
- **测试场景**:
 - **准确率评估**: 验证 LLM 是否能正确识别实体、指标与筛选条件（例如：将“北京”映射为 `filters: [{id: DIM_CITY, val: "Beijing"}]`）。
 - **Prompt 调试**: 开发人员可高频调用此接口调整 Prompt，无需建立真实的数据库连接。

5.2.2 编译器接口 (The "Compiler" Check)

- **Path**: `POST /nl2sql/sql`
- **覆盖范围**: Stage 4 (SQL 生成)
- **输入**: `Validated Plan` (JSON)
 - *注：支持直接构造或修改 JSON 作为入参，实现对前置 LLM 阶段的 Mock。*
- **输出**: `SQL String` (Target Dialect)
- **测试场景**:
 - **确定性逻辑回归**: 此阶段不涉及 LLM，输入 A 必然得到输出 B。适合编写单元测试验证 SQL 语法正确性、方言适配及安全注入逻辑。
 - **边界条件测试**: 测试人员可手动构造复杂的 Plan（如包含多层嵌套或特殊字符），验证 SQL 生成器的健壮性。

5.2.3 执行接口 (End-to-End)

- **Path**: `POST /nl2sql/execute`
- **覆盖范围**: Stage 1 ~ Stage 6 全流程
- **输入**: 自然语言问题 (String) + 用户上下文 (Context)

- **输出:** `FinalAnswer` (Text + Data)
- **测试场景:**
 - **用户验收测试 (UAT):** 前端应用、ChatFlow 或业务人员使用的主要入口。
 - **全链路验证:** 确保生成的 SQL 在真实数据库中执行无误，且数据权限控制生效。

5.3 全链路追踪与调试 (Tracing & Debugging)

为了支持线上排查与复杂调试，系统具备**“白盒化”**能力。核心接口支持通过参数透出流水线内部的中间产物，实现全链路可观测。

5.3.1 调试模式 (Debug Mode via Request Flag)

接口（特别是 `/nl2sql/execute`）支持可选参数 `include_trace` (Boolean)。

- **行为定义:**
 - `include_trace=false` (默认): 仅返回最终的 `FinalAnswer`，保持响应体轻量。
 - `include_trace=true`: 在响应体的 `debug_info` 字段中，完整透出 Stage 1 ~ Stage 5 的中间产物。
- **应用价值:**
 - 当测试人员在 UI 发现回答错误时，勾选此选项即可立即获取对应的 Plan 和 SQL，无需通过服务器后台日志抓取 Request ID 进行二次排查。

5.3.2 响应结构规范 (Trace Response Schema)

当启用调试模式时，API 响应结构扩展如下：

代码块

```
1  {
2    "status": "SUCCESS",
3    "data": {
4      // Stage 6 的最终产物 (FinalAnswer)"answer_text": "...",
5      "data": [...]
6    },
7    "debug_info": {
8      // [新增] 全链路追踪数据，仅在 include_trace=true 时返回"stage1_subqueries":
9      [ ... ],
10     "stage2_raw_plan": { ... },
11     "stage3_validated_plan": { ... },
12     "stage4_final_sql": "SELECT ...",
13     "stage5_meta": {
14       "latency_ms": 120,
15       "row_count": 50,
16       "is_truncated": false
17     }
18   }
```

```
17     }
18 }
```

结合具体的场景举个例子，如下：

示例背景 (Context)

- **用户提问：**“帮我看下**华北区**最近**30天**的**销售额趋势**，只看**自营渠道**的。”
- **当前时间:** 2023-11-01
- **租户 ID:** `tenant_888`
- **请求参数:** `include_trace=true`
- **响应结构 (Trace Response Schema)**如下：

代码块

```
1  {
2      //
   =====
   ===
3      // 1. 标准业务响应 (Standard Response)
4      // 这里的结构完全符合 Stage 6 的输出契约，前端/ChatFlow 主要消费这部分。
5      //
   =====
   ===
6      "status": "SUCCESS",
7      "request_id": "req_20231101_af92d8",
8      "data": {
9          "status": "SUCCESS", // Stage 6 内部状态 (可能为 CLARIFICATION_NEEDED)
10
11         // [Stage 6产物] LLM 生成的自然语言分析
12         "answer_text": "过去30天（10月1日至10月30日），华北区自营渠道的销售额整体呈上升趋势。最高峰出现在 **10月25日**，当日销售额为 **15.2万元**。建议关注月底的库存补给情况。",
13
14         // [Stage 5产物] 清洗后的前端可用数据
15         "data": {
16             "columns": [
17                 {"name": "ORDER_DATE", "type": "STRING", "display_name": "下单日期"},
18                 {"name": "GMV", "type": "FLOAT", "display_name": "销售额"}
19             ],
20             "rows": [
21                 ["2023-10-01", 12000.50],
22                 ["2023-10-02", 11500.00],
23                 // ... 中间省略数据 ...
24                 ["2023-10-30", 14800.20]
25             ],
```

```

26     "is_truncated": false // 未触发最大行数限制
27 },
28
29     // 错误信息 (成功时为 null)
30     "error": null
31 },
32
33     //
=====
===
34     // 2. 全链路调试信息 (Debug Trace Info)
35     // 仅在请求参数 include_trace=true 时返回。
36     // 用于测试人员排查：是 LLM 理解错了？还是 SQL 生成错了？还是数据库慢了？
37     //
=====
===
38     "debug_info": {
39
40         // [Stage 1 Output] 意图拆解
41         // 观察 LLM 是否正确理解了复杂句式，将问题拆解为原子查询
42         "stage1_subqueries": [
43             {
44                 "id": "q1",
45                 "description": "统计华北区、自营渠道在过去30天内，按日期的销售额(GMV)趋势"
46             }
47         ],
48
49         // [Stage 2 Output] 原始 Plan (LLM 生成)
50         // 观察 LLM 的“幻觉”情况。重点检查：Filter 是否提取正确？Intent 是否为 TREND？
51         "stage2_raw_plan": {
52             "intent": "TREND",
53             "metrics": ["METRIC_GMV"],
54             "dimensions": ["DIM_ORDER_DATE"], // LLM 自动推断出需要按日期分组
55             "filters": [
56                 {"id": "DIM_REGION", "op": "EQ", "values": ["华北"]},
57                 {"id": "DIM_CHANNEL", "op": "EQ", "values": ["自营"]}
58             ],
59             "time_range": {
60                 "type": "LAST_N",
61                 "value": 30,
62                 "unit": "DAY"
63             }
64         },
65
66         // [Stage 3 Output] 校验后 Plan (逻辑补全)
67         // 观察后端逻辑的介入：
68         // 1. time_range 是否转成了绝对时间？

```

```

69 // 2. limit 是否补全?
70 // 3. ID 是否经过了存在性校验?
71 "stage3_validated_plan": {
72     "intent": "TREND",
73     "metrics": [
74         {"id": "METRIC_GMV", "agg": "SUM", "field": "amount"} // 补全了元数据
75     ],
76     "dimensions": [
77         {"id": "DIM_ORDER_DATE", "field": "created_at", "time_grain": "DAY"}
78     ],
79     "filters": [
80         {"id": "DIM_REGION", "op": "EQ", "values": ["North China"]}, // 假设内部
做了值映射
81         {"id": "DIM_CHANNEL", "op": "EQ", "values": ["Self-Operated"]}
82     ],
83     "time_range": {
84         "type": "ABSOLUTE",
85         "start": "2023-10-01",
86         "end": "2023-10-30",
87         "original_text": "Last 30 Days"
88     },
89     "limit": 100 // 默认注入的 Limit
90 },
91
92 // [Stage 4 Output] 最终 SQL
93 // 观察 PyPika 的翻译能力 & 安全注入:
94 // 1. DATE_FORMAT 函数是否符合当前数据库方言?
95 // 2. tenant_id 是否强制注入了? (最关键的安全检查点)
96 "stage4_final_sql": "SELECT DATE_FORMAT(t1.created_at, '%Y-%m-%d') AS
DIM_ORDER_DATE, SUM(t1.amount) AS METRIC_GMV FROM fact_orders t1 LEFT JOIN
dim_region t2 ON t1.region_id = t2.id WHERE t1.created_at >= '2023-10-01' AND
t1.created_at <= '2023-10-30' AND t2.name = 'North China' AND t1.channel =
'Self-Operated' AND t1.tenant_id = 'tenant_888' GROUP BY
DATE_FORMAT(t1.created_at, '%Y-%m-%d') LIMIT 100",
97
98 // [Stage 5 Output] 执行元数据
99 // 观察性能指标:
100 // 1. SQL 执行慢不慢? (latency_ms)
101 // 2. 结果集大不大? (row_count)
102 "stage5_meta": {
103     "executed_at": "2023-11-01T12:00:05Z",
104     "latency_ms": 145.5, // 毫秒
105     "row_count": 30,
106     "db_engine": "mysql" // 标识当前使用的底层适配器
107 }
108 }
109 }

```

6. 附录 (Appendices)

附录 A：提示词与文案模板库 (Prompts & Templates Registry)

本附录集中管理系统中所有面向大模型（LLM）的 **提示词 (Prompts)** 和面向最终用户的 **静态回复模板 (Static Templates)**。

在工程实现中，这些常量应统一托管于 `nl2sql_service/utils/prompt_templates.py` 文件中，严禁在业务代码中硬编码字符串。

A.1 动态提示词 (Dynamic LLM Prompts)

此类提示词包含变量占位符（`{var}`），需在运行时注入上下文后调用 LLM 生成。

1. 意图拆解 (Stage 1)

- 常量名:** `PROMPT_SUBQUERY_DECOMPOSITION`
- 用途:** 将用户复杂问题拆解为原子子查询。
- 输入变量:** `{current_date}`，`{question}`
- 配置:**
 - `temperature` : `0.0` (确保拆解逻辑的绝对确定性)
 - `response_format` : `{"type": "json_object"}` (强制 JSON 输出)
- 定义:**

代码块

```
1  PROMPT_SUBQUERY_DECOMPOSITION = """
2  你是一个“查询需求拆解器”，用于 NL2SQL 场景的前置处理。
3
4  【你的任务】
5  给定用户提出的一句话业务问题（question），请将这个问题拆解为 不超过3条“子查询”（sub_queries），用自然语言精确描述每条子查询的统计需求。
6  这些子查询会被下游 NL → PLAN 模块进一步转换为结构化查询计划。
7
8  【拆解原则】
9  1. 永远至少生成 1 条子查询：
10     - 如果原问题比较笼统，只生成 1 条描述更清晰的“规范化问题”。
11  2. 如果原问题明显包含多个意图，可以拆成多条子查询：
12     - 并列意图：
```

- 原文：“看一下 GMV 趋势，并且告诉我 TOP3 客户”
- 拆解：“统计最近一段时间每天的 GMV 趋势”、“统计 GMV 排名前 3 的客户”。
- 时间或人群对比：
 - 原文：“今年和去年华南大区的销售额对比一下”
 - 拆解：“统计今年华南大区的销售额”、“统计去年华南大区的销售额”。
- 3. 拆解数量控制：
 - 一般不超过 3 条子查询，除非原问题非常明确包含更多独立子问题。
- 4. 子查询的描述 (description) 要求：

核心原则：生成的 description 必须是“独立、完整、机器可读”的业务指令。

要求1：结构化表达 (Structure)

 - 严格遵循格式：[绝对时间范围] + [分组维度(可选)] + [统计指标]。
 - 动词统一：使用“统计”、“查询”或“计算”开头。



要求2：时间绝对化 (Time Resolution)

 - 必须将所有相对时间词（如“今年”、“上个月”）转换为具体的日期范围。
 - 依据当前日期：{current_date}（请以此日期为基准进行推算）。
 - 转换规则：“今年” → “2023年”；“近3天” → “2023-10-01 至 2023-10-03”。

要求3：独立完整性 (De-contextualization)

 - 补全省略的主语，消除上下文依赖。
 - 示例：原句“利润率呢？” → 拆解“统计 2023年 华南区的利润率”。

要求4：严禁指标脑补 (Anti-Hallucination) — CRITICAL

 - 保持原意：如果用户使用模糊词汇（如“业绩”、“成效”），请在 description 中保留该模糊词，严禁擅自将其细化为具体指标（如将“业绩”脑补为“员工业绩”或“GMV”）。
 - 理由：模糊词的消歧义工作将由下游模块完成，不需要你在此阶段猜测。
 - 示例：
 - 原句：“查看今年的业绩”
 -  “统计 2023年 的业绩”（保留原词）
 -  “统计 2023年 的员工业绩”（禁止脑补）
- 5. 不要做的事情：
 - 不要输出 SQL；
 - 不要输出解释性文字或分析结论；
 - 不要输出和 schema 无关的字段；
 - 不要在 JSON 外再加任何说明文字。

【输出格式】

你必须输出一个合法的 JSON 数组，顶层就是子查询对象列表。

数组中每个元素的结构如下：

```
[
  {
    "id": "q1",          // 建议使用 "q1", "q2" 等简短标识
    "description": "string" // 子查询的中文需求描述
  }
]
```

```
59 具体要求：
60 1. 顶层是一个 JSON 数组，不能再包一层 { "sub_queries": [...] }。
61 2. 数组必须非空，至少包含 1 个子查询对象。
62 3. description 不得为空字符串，建议长度在 10~60 字之间。
63
64 【兜底逻辑】
65 如果你无法合理拆分（例如问题非常含糊），请按以下方式返回：
66 - 仍然只返回 1 条子查询，id 为 "q1"。
67 - description = 对原始 question 略微规范化后的版本。
68 如果用户输入的是乱码或完全无关的话（“今天天气不错”）：
69 - 仍然只返回 1 条子查询，id 为 "q1"。
70 - description = "INVALID_QUERY"
71
72 【用户输入】
73 当前日期：{current_date}
74 用户问题："{question}"
75
76 【JSON 输出】
77 ""
```

2. Plan 生成 (Stage 2)

- **常量名:** PROMPT_PLAN_GENERATION
- **用途:** 将自然语言转换为结构化 Query Plan (JSON)。
- **输入变量:** {current_date}, {user_query}, {schema_context}
- **配置:**
 - temperature : 0.0 (Schema 映射严禁任何随机性)
 - response_format : {"type": "json_object"} (强制 JSON 输出)
- **定义:**

代码块

```
1  # Role
2  You are an expert Data Analyst AI. Your goal is to map the user's natural
   language question into a structured **JSON Query Plan**.
3
4  # Context
5  - Current Date: {current_date} (Format: YYYY-MM-DD)
6  - User Question: "{user_query}"
7
8  # Available Schema (Retrieved Context)
9  You can ONLY use the Metrics and Dimensions listed below.
10 **CRITICAL RULE:** Do NOT invent new IDs. Do NOT use IDs that are not in this
    list.
```

```

11 -----
12 {schema_context}
13 -----
14
15 # Logic Rules & Constraints
16
17 1. Intent Classification:
18     - `AGG`: If the user asks for aggregated numbers (e.g., "Total Sales",
19       "Count orders").
20     - `TREND`: If the user asks for trends over time (e.g., "Monthly sales
21       trend", "Daily active users").
22     - `DETAIL`: If the user asks for raw records (e.g., "List recent orders",
23       "Show employee details").
24
25 2. Metrics & Dimensions:
26     - Map user phrases to the exact `ID` from the Available Schema.
27     - Metrics:
28     - If user asks for comparison (e.g., "YoY", "同比"), set
29       `compare_mode`="YOY".
30     - If "MoM"/"环比", set `compare_mode`="MOM".
31     - If "WoW"/"周环比", set `compare_mode`="WOW".
32     - Otherwise, set `compare_mode`=null.
33     - Dimensions:
34     - CRITICAL: Only set `time_grain` (e.g., "DAY", "MONTH") if the
35       dimension is marked with `| Is_Time: True` in the Schema Context.
36     - For all other dimensions, `time_grain` MUST be null.
37
38 3. Filters:
39     - Put ALL filter conditions here (do not distinguish between WHERE and
40       HAVING).
41     - `id`: Must be a valid Metric ID or Dimension ID.
42     - `op`: Choose from ["EQ", "NEQ", "IN", "NOT_IN", "GT", "LT", "GTE", "LTE",
43       "BETWEEN", "LIKE"].
44     - `values`: Must be a list. Strictly keep original types. Do NOT
45       convert string IDs (e.g., "007") to numbers (7).
46
47 4. Time Range:
48     - Relative: If user says "last 7 days", use `{"type": "LAST_N",
49       "value": 7, "unit": "DAY"}`.
50     - Absolute: If user says "2023-01-01 to 2023-01-31", use `{"type":
51       "ABSOLUTE", "start": "2023-01-01", "end": "2023-01-31"}`.
52     - Missing: If user mentions NO time, set `time_range` to `null`. (Do
53       NOT guess a default time).
54
55 5. Order By & Limit:
56     - `order_by`: Only sort by metrics or dimensions that are selected in the
57       query.

```

```

46     - `limit`: Set to integer if user specifies (e.g., "Top 10"). Otherwise
      `null`.
47
48   # JSON Output Format
49   Return ONLY a valid JSON object matching this structure:
50
51   ```json
52   {
53     "intent": "AGG",
54     "metrics": [
55       { "id": "METRIC_ID", "compare_mode": "YOY" }
56     ],
57     "dimensions": [
58       { "id": "DIM_ID", "time_grain": "MONTH" }
59     ],
60     "filters": [
61       { "id": "DIM_ID", "op": "EQ", "values": ["Value"] }
62     ],
63     "time_range": {
64       "type": "LAST_N",
65       "value": 30,
66       "unit": "DAY",
67       "start": null,
68       "end": null
69     },
70     "order_by": [
71       { "id": "METRIC_ID", "direction": "DESC" }
72     ],
73     "limit": 100
74   }

```

3. 数据洞察分析 (Stage 6 - Success Path)

- **常量名:** PROMPT_DATA_INSIGHT
- **用途:** 基于查询结果生成自然语言总结。
- **输入变量:** {user_query}, {query_context}, {data_markdown}
- **配置:** temperature`: **0.2** (增加少量语言润色, 避免过度机械, 但严防数值幻觉)
- **定义:**

代码块

```

1  PROMPT_DATA_INSIGHT = """
2  你是一位专业的商业数据分析师。请根据用户的问题、查询意图摘要和下方提供的数据结果（可能包含
   多个子查询的数据表），生成一份综合性的数据综述。
3

```

```
4    【用户问题】
5    {user_query}
6
7    【查询意图摘要】
8    {plan_summary}
9
10   【数据结果】
11   {multi_table_markdown}
12
13   【回答要求】
14   1. **分别陈述 (Critical)**: 请逐一解读每个子查询的数据表。**严禁** 自行对不同表格的数据
      进行数学运算（如将表A的数值除以表B的数值），防止产生计算幻觉。
15   2. **结论先行**: 直接回答用户问题。
16   3. **处理缺失**: 如果数据结果中包含“部分数据缺失”的提示，请在回答末尾礼貌告知用户。
17   4. **客观陈述**: 仅描述表格中存在的趋势、极值或异常点。严禁编造数据。
18   5. **简洁明了**: 篇幅控制在 150 字以内。
19   """
```

4. 意图澄清追问 (Stage 6 - Clarification Path)

- 常量名: `PROMPT_CLARIFICATION`
- 用途: 当出现歧义错误时，生成礼貌追问。
- 输入变量: `{user_query}` , `{error_message}`
- 配置: `temperature: 0.3` (允许语气更加柔和、礼貌，提升交互体验)
- 定义:

代码块

```
1    PROMPT_CLARIFICATION = """
2    你是一个智能数据助手。用户的问题因为“语义模糊”导致无法执行，系统返回了具体的错误原因。
3    请根据错误原因，生成一句礼貌的追问，引导用户澄清意图。
4
5    【用户问题】
6    {user_query}
7
8    【系统错误原因】
9    {error_message}
10
11   【生成要求】
12   1. 语气礼貌、乐于助人。
13   2. 将技术性的错误原因转化为用户能听懂的语言。
14   3. 给出具体的选项供用户参考（如果错误信息里包含了选项）。
15   4. **不要** 暴露系统内部错误码（如 AMBIGUOUS_TIME）。
16
17   【生成示例】
```

```
18  错误原因: Found multiple metrics for 'revenue': ['gross_revenue', 'net_revenue']
19  你的回答: 我找到了两个相关的指标: **毛利**和**净利**。请问您具体想看哪一个?
20  """"
```

A.2 静态文案模板 (Static Response Templates)

此类模板不需要调用 LLM，通过 Python 字符串格式化 (f-string) 填充变量，通常用于权限拒绝或系统错误的兜底响应。

5. 拒绝类文案

- **常量名:** `TEMPLATE_REFUSAL`
- **用途:** 权限不足、跨事实表查询等被系统拦截的场景。
- **输入变量:** `{error_msg}` (来自 Stage 3/5 产生的友好报错信息)
- **定义:**

代码块

```
1  TEMPLATE_REFUSAL = (  
2      "抱歉，无法执行该查询。\\n""原因: {error_msg}。\\n""建议您调整提问方式或联系管理员。"  
3  )
```

6. 系统故障文案

- **常量名:** `TEMPLATE_SYSTEM_ERROR`
- **用途:** 数据库连接失败、LLM 超时等不可恢复的系统级错误。
- **输入变量:** 无
- **定义:**

代码块

```
1  TEMPLATE_SYSTEM_ERROR = (  
2      "系统似乎开小差了（服务暂时繁忙）。""技术团队正在抢修中，请您稍后再试。"  
3  )
```

附录B 请求/响应示例 (Sample Requests & Responses)

- 展示典型 `/execute` 请求与响应 JSON。

附录 C：API 异常响应协议（API Error Response Protocol）

本附录定义了系统在触发 **第 10 章：异常处理与流转机制** 中定义的“澄清”或“错误”状态时，对外输出的标准 JSON 结构。

C.1 统一响应结构 (Response Schema)

代码块

```
1  {
2    "status": "ERROR", // 或 "NEED_CLARIFICATION""request_id":
    "req_20250310_00001",
3    "error": {
4      "stage": "STAGE_3_VALIDATION",
5      "code": "AMBIGUOUS_INTENT",
6      "message": "检测到指标歧义，无法确定具体统计对象。",
7      "data": {
8        "candidates": ["个人业绩", "部门业绩"],
9        "recommendation": "请明确是指个人还是部门。"
10     }
11   }
12 }
```

C.2 字段详细说明

字段	类型	说明
status	string	响应状态标记。 · ERROR: 对应第三级错误（系统故障/非法请求），HTTP 4xx/5xx。 · NEED_CLARIFICATION: 对应第二级澄清（歧义），HTTP 200。
error.stage	string	出错阶段。 枚举：STAGE_1_ROUTER, STAGE_2_PLANNER, STAGE_3_VALIDATOR, STAGE_4_COMPILER, STAGE_5_EXECUTOR。
error.code	string	机器错误码。用于前端多语言映射或特定逻辑处理（如 AMBIGUOUS_TIME 触发时间选择器）。
error.message	string	人类可读描述。默认的兜底文案，用于调试或直接展示。
error.data	object	结构化上下文（可选）。用于“澄清”场景，携带候选集（candidates）、冲突详情等，供前端或 Stage 6 生成交互式追问。

C.3 常用错误码映射表 (Error Code Mapping)

错误分类	Error Code	触发 Stage	HTTP 状态	含义
澄清类	AMBIGUOUS_INTENT	Stage 3	200	意图/指标/维度存在多义性
	AMBIGUOUS_TIME	Stage 3	200	默认时间窗口冲突
	MISSING_METRIC	Stage 3	200	聚合查询未指定指标
错误类	PERMISSION_DENIED	Stage 2/3	403	越权访问
	UNSUPPORTED_MULTI_FACT	Stage 3	400	跨事实表查询（MVP 不支持）
	INVALID_PLAN_STRUCTURE	Stage 2	500	LLM 生成格式严重错误
	SQL_EXECUTION_TIMEOUT	Stage 5	504	数据库查询超时

附录D 全局配置

为了方便系统调教（Tuning）并保持代码硬编码最小化，本项目的关键超参数统一托管在全局配置类 (PipelineConfig)中,, 它是一个简单的配置类，在项目根目录 config.py中定义

在本文中，需要设置超参数的地方，统一描述为“调参锚点”，可以在文中搜索关键字“调参锚点”快速查看所有的具体的超参数的具体位置，定义，初值，后续会统一管理在**全局配置类 (PipelineConfig)**中。

这个配置类（PipelineConfig）的简单示例如下：

代码块

```
1  from enum import Enum
2  from pydantic import BaseModel, Field
3
4  class SupportedDialects(str, Enum):
5      """
6      MVP 阶段仅支持 MySQL 和 PostgreSQL
7      """
8      MYSQL = "mysql"
9      POSTGRESQL = "postgresql"
10
11  class PipelineConfig(BaseModel):
12      """
13      NL2SQL 流水线全局配置类
14      集中管理各阶段的策略阈值、限制参数与开关
15      """
16
17      # ===== Stage 2: 检索与生成 (Retrieval & Generation)
18      =====
```

```
19     # [调参锚点 1] 向量检索召回数量 (Vector Search Top-K)
20     # 定义：在向量数据库中初步检索出的候选 Term 数量。
21     # 建议：20-30。过少易漏关键指标，过多会增加后续处理开销。
22     vector_search_top_k: int = Field(default=20, ge=1, le=100, description="向量
检索阶段的 Top-K")
23
24     # [调参锚点 2] 最终召回上限 (Max Term Recall)
25     # 定义：经过精确匹配混合、去重、截断后，最终喂给 LLM Prompt 的 Term 最大数量。
26     # 建议：20-50。受限于 LLM Context 窗口和注意力分散问题，不宜过多。
27     max_term_recall: int = Field(default=20, ge=1, le=50, description="注入
Prompt 的最大 Term 数量")
28
29     # [调参锚点 3] 相似度截断阈值 (Similarity Threshold)
30     # 定义：向量检索分数的最低门槛。低于此分数的 Term 被视为不相关，直接丢弃，不进入
Prompt。
31     # 建议：0.4-0.6。需根据 Embedding 模型表现微调。
32     similarity_threshold: float = Field(default=0.4, ge=0.0, le=1.0,
description="向量相似度过滤阈值")
33
34     # [调参锚点 4] Schema 描述最大长度 (Max Description Length)
35     # 定义：在 Prompt 的 Schema Context 中，对每个指标/维度描述文本的字符截断长度。
36     # 建议：50-100。防止单个指标的长篇大论挤占 Token。
37     max_description_length: int = Field(default=50, ge=10, le=200,
description="Schema 描述的字符截断长度")
38
39     # [调参锚点 5] 枚举值展示数量 (Max Enum Values Display)
40     # 定义：在 Prompt 中，为每个维度展示的枚举值示例数量（防止幻觉）。
41     # 建议：5-10。
42     max_enum_values_display: int = Field(default=8, ge=1, le=20,
description="Prompt 中展示的枚举值示例数量")
43
44
45     # ===== Stage 3: 校验与补全 (Validation & Normalization)
=====
46
47     # [调参锚点 6] 默认查询行数 (Default Limit)
48     # 定义：当用户未指定 "Top N" 时，Plan 中自动填充的 limit 值。
49     # 建议：100。
50     default_limit: int = Field(default=100, ge=1, description="用户未指定时的默
认 Limit")
51
52     # [调参锚点 7] 最大查询行数上限 (Max Limit Cap)
53     # 定义：即使用户指定了 limit，也不允许超过此值（防止恶意拉取全表）。
54     # 建议：1000-5000。
55     max_limit_cap: int = Field(default=1000, ge=100, description="允许的最大
Limit 上限")
56
```

```

57     # [调参锚点 8] 默认回溯天数 (Default Lookback Days)
58     # 定义：当用户意图为 TREND/AGG 但未指定时间范围时，默认查询过去多少天的数据。
59     # 建议：30（月度视角）或 7（周视角）。
60     default_lookback_days: int = Field(default=30, ge=1, description="默认时间窗口回溯天数")
61
62
63     # ===== Stage 4 & 5: 生成与执行 (Generation & Execution)
64     =====
65     # [调参锚点 9] 数据库方言 (Database Dialect)
66     # 定义：目标数据库类型，决定了 SQL 生成函数 (Stage 4) 和会话防御指令 (Stage 5)。
67     # 约束：MVP 仅支持 mysql 或 postgresql。
68     db_type: SupportedDialects = Field(default=SupportedDialects.POSTGRESQL,
69                                         description="目标数据库方言")
70
71     # [调参锚点 10] SQL 执行超时时间 (Execution Timeout)
72     # 定义：数据库会话的强制超时时间（毫秒）。
73     # 建议：5000ms（5秒）。OLAP 查询通常较慢，但作为 API 服务不应超过 10秒。
74     execution_timeout_ms: int = Field(default=5000, ge=1000, le=60000,
75                                       description="SQL 执行超时时间(ms)")
76
77     # [调参锚点 11] 结果集硬截断行数 (Max Result Rows / Hard Limit)
78     # 定义：后端从数据库 Fetch 数据的最大物理行数，防止 OOM。
79     # 建议：5000。注意：此值应 >= max_limit_cap。
80     max_result_rows: int = Field(default=5000, ge=1000, description="结果集物理截断行数")
81
82     # 全局单例配置实例
83     pipeline_config = PipelineConfig()

```

附录E 名词与缩写 (Terminology)

为避免歧义，本规范中使用的关键术语与缩写如下（与《PLAN 模型：结构定义与兜底策略》《语义层配置与 YAML Schema 规范》保持一致）：

- **NL (Natural Language)**

用户输入的自然语言问题，例如：“最近 30 天华南大区的 GMV 趋势怎么样？”

- **subquery (子问题)**

从复杂问题中拆解出的、可以独立回答的子问题。

例如：“最近 30 天华南大区的 GMV 趋势”和“与上月同期对比”可以拆成两个 subqueries。

- **PLAN（查询计划 / Query Plan）**

由 NL 解析得到的、结构化的查询计划对象，描述“查什么数、按什么维度、在什么时间范围、带什么过滤”，是 NL 与 SQL 之间的中间表示。

具体字段与约束详见《PLAN 模型：结构定义与兜底策略》。

- **semantic view（语义视图）**

建立在底层事实表 / 维表之上的视图（如 `v_sales_order_item`、`v_employee_profile`），按业务实体进行预建模，对外通过语义层暴露，隐藏底层存储细节。

- **RLS（Row-Level Security，行级权限）**

按行限制用户可见的数据范围，例如：

- 员工只能看自己相关记录；
- 部门负责人只能看本部门数据；
- 区域负责人只能看负责区域的数据。

- **CLS（Column-Level Security，列级权限）**

按列限制用户可见的字段范围，例如：

- 普通用户不可见成本、利润、薪资等敏感字段；
- 对部分字段进行脱敏（如隐藏个人联系方式的部分内容）。

- **Pipeline Stage（流水线阶段）**

在本规范中，端到端流程被拆分为多个职责单一的阶段（Stage），每个阶段有明确的输入、输出和错误行为。

- **Execution Result（Result）**

数据库执行 SQL 后返回的结构化结果（行集 + 列信息），在 Answer 阶段之前仍保持“数据”语义，不包含自然语言解释。

附录F 关键类和算法说明

1、SemanticRegistry

完整定义：

代码块

```
1 from typing import Dict, List, Optional, Any, Set
2 from pydantic import BaseModel
3
4 # 假设 SchemaDef 是所有定义类的基类 (MetricDef, DimensionDef, etc.)
5 class SchemaDef(BaseModel):
```

```

6     id: str
7     type: str # METRIC, DIMENSION, ENTITY, etc.
8     # ... 其他共有字段
9
10 class SemanticRegistry:
11     """
12     语义层核心注册表 (Singleton)
13     职责: 加载 YAML 配置, 维护内存索引, 提供元数据查询与检索服务。
14     """
15
16     def __init__(self):
17         # =====
18         # 1. 核心数据存储 (Memory Storage)
19         # =====
20
21         # [全量元数据]
22         # Key: ID (e.g., "METRIC_GMV"), Value: 定义对象
23         # 来源: semantic_metrics.yaml, semantic_core.yaml
24         self.metadata_map: Dict[str, SchemaDef] = {}
25
26         # [倒排索引]
27         # Key: 中文别名/名称 (e.g., "销售额"), Value: ID 列表
28         # 用途: Stage 2 精确匹配
29         self.keyword_index: Dict[str, List[str]] = {}
30
31         # [全局配置]
32         # Key: 配置项 (e.g., "default_time_window"), Value: 值
33         # 来源: semantic_common.yaml
34         self.global_config: Dict[str, Any] = {}
35
36         # [安全策略]
37         # 结构: 存储 semantic_security.yaml 解析后的策略对象
38         # 用途: 权限过滤与 RLS
39         self._security_policies: List[Any] = []
40
41         # =====
42         # 2. 外部服务客户端 (External Clients)
43         # =====
44         self.qdrant_client = AsyncQdrantClient(...)
45         self.jina_client = AsyncJinaClient(...)
46
47         # =====
48         # A. 初始化与加载 (Initialization)
49         # =====
50     async def load_from_yaml(self, yaml_path: str = "semantics/"):
51         """
52         启动时调用。

```

```

53         1. 读取 YAML -> 填充 metadata_map, keyword_index, global_config,
        _security_policies
54         2. 计算指纹 -> 检查 Qdrant 状态 -> (必要时) 调用 Jina 重建向量索引
55         """
56         pass
57
58     # =====
59     # B. 基础查询 (Basic Lookup)
60     # =====
61     def get_term(self, term_id: str) -> Optional[SchemaDef]:
62         """通用获取方法, 返回 Metric/Dim/Entity 等对象"""
63         return self.metadata_map.get(term_id)
64
65     def get_metric_def(self, metric_id: str) -> Optional[Any]:
66         """获取指标定义的强类型封装 (Stage 3/4 常用)"""
67         obj = self.get_term(metric_id)
68         return obj if obj and obj.type == 'METRIC' else None
69
70     def get_entity_def(self, entity_id: str) -> Optional[Any]:
71         """获取实体定义的强类型封装 (Stage 4 常用)"""
72         obj = self.get_term(entity_id)
73         return obj if obj and obj.type == 'ENTITY' else None
74
75     # =====
76     # C. 智能检索 (Search & Retrieval)
77     # =====
78     async def search_similar_terms(self, query: str, allowed_ids: List[str],
    top_k: int = 20) -> List[tuple]:
79         """
80         Stage 2 核心方法。
81         1. 调用 Jina Embedding
82         2. 调用 Qdrant Search (带 allowed_ids 过滤)
83         3. 返回 [(id, score), ...]
84         """
85         pass
86
87     # =====
88     # D. 业务逻辑校验 (Validation Logic)
89     # =====
90     def check_compatibility(self, metric_id: str, dimension_id: str) -> bool:
91         """
92         Stage 3 核心方法。
93         判断指标和维度是否属于同一个 Entity (语义视图)。
94         """
95         m_def = self.get_metric_def(metric_id)
96         d_def = self.get_term(dimension_id) # Dimension 可能在 Core 中
97

```

```

98         if not m_def or not d_def:
99             return False
100
101         # 核心逻辑: 比较 Entity ID 是否一致
102         return m_def.entity_id == d_def.entity_id
103
104     # =====
105     # E. 安全与权限 (Security)
106     # =====
107     def get_allowed_ids(self, role_id: str) -> Set[str]:
108         """
109         Stage 2/3 核心方法。
110         根据 semantic_security.yaml 计算该角色有权访问的所有 ID 集合。
111         """
112         pass
113
114     def get_rls_policies(self, role_id: str, entity_id: str) -> List[str]:
115         """
116         Stage 4 核心方法。
117         获取该角色针对特定 Entity 的 RLS 过滤条件 (SQL 片段)。
118         """
119         pass

```

该类在全局调用的地方如下：

阶段	场景	需求描述	对应方法/属性
Start	启动初始化	加载 YAML，计算指纹，初始化 Qdrant/Jina，构建内存索引。	load_from_yaml()
Stage 2	权限预过滤	根据 role_id 获取该用户能看到的所有 Metric/Dim ID 白名单。	get_allowed_ids(role_id)
Stage 2	关键词检索	根据用户 Query 中的关键词（如“营收”）查找精确匹配的 ID。	keyword_index (属性)
Stage 2	向量检索	调用 Embedding 和 Qdrant 查找语义相似的 ID。	search_similar_terms(query, allowed_ids)
Stage 2	枚举透出	获取某个维度关联的枚举值列表（用于 Prompt）。	get_enum_def(enum_id)
Stage 3	存在性校验	检查 PLAN 中的 ID 是否真实存在。	get_term(id) / exists(id)
Stage 3	兼容性校验	检查指标和维度是否属于同一个 Entity (语义视图)。	check_compatibility(metric_id, dim_id)
Stage 3	默认值补全	获取指标的 default_time，或全局 default_time_window。	get_metric_def(id), global_config
Stage 3	强制过滤	获取指标绑定的 mandatory_filters。	get_metric_def(id)
Stage 4	物理映射	获取 Entity 对应的物理视图名，Metric 对应的 SQL 表达式。	get_entity_def(id), get_metric_def(id)
Stage 4	RLS 注入	根据 role_id 获取行级权限 SQL 片段。	get_qls_policies(role_id)