

语义层配置规范

1. 概述与设计哲学

1.1 来源与定位

本项目的 YAML 配置体系是连接 **业务逻辑** 与 **技术实现** 的桥梁。

- **来源**：源于《业务术语表》、《指标口径文档》及《数据仓库 DDL》。
- **定位**：它是 NL2SQL 系统的“静态大脑”。
 - **对LLM (Stage 2)**：它是 **知识库 (Knowledge Base)**。它告诉 LLM 业务域里有哪些指标、维度，以及它们的自然语言别名是什么。
 - **对后端(Stage 3和4)**：它是 **规则引擎 (Rule Engine)**。它告诉代码如何校验参数合法性、如何拼接物理 SQL、如何注入安全策略。

1.2 核心设计原则

为了确保配置的可维护性与系统的高效运行，本规范遵循以下三大铁律：

1. 高内聚 (High Cohesion)

- 遵循 "Definition & Alias Locality" 原则。
- 谁定义了对象，谁就负责管理它的别名 (Aliases) 和枚举值 (Enums) 。
- **反例**：严禁将“指标定义”放在一个文件，而将“指标同义词”散落在另一个独立的同义词典文件中。

2. RAG 友好 (RAG-Native)

- 文件结构设计天然适配向量索引构建。
- **Chunking 友好**：单个 YAML 节点（如一个 Metric 对象）包含了构建 Embedding 所需的全部上下文 (ID + Name + Description + Aliases) 。
- **无需 Join**：索引构建程序无需跨文件关联即可生成完整的语义向量。

3. 单一事实来源 (Single Source of Truth)

- 所有的指标口径、维度映射、权限规则、默认值策略，**必须且只能**在 YAML 中定义。
- **严禁**在 Python 代码中硬编码业务逻辑（如 `if metric == 'GMV': sql = 'sum(amount)'`）。代码只负责“读取配置并执行”，不负责“定义规则”。

1.3 YAML职责和消费链路 (Consumption Mapping)

为了帮助开发人员理解配置文件的上下游关系，本节详细列出了每个核心配置文件在流水线各阶段的具体消费场景与作用。

1. 指标定义配置 (semantic_metrics.yaml)

- **核心职责：**定义指标的计算公式、默认时间窗口、同义词等业务属性。
- **消费链路：**
 - **Stage 2 (RAG 检索)：**系统读取指标的名称与别名构建向量索引，用于根据用户自然语言召回对应的 METRIC_ID。
 - **Stage 3 (校验与补全)：**提供 default_time（默认时间）和 mandatory_filters（强制过滤），用于在 Plan 不完整时进行默认值补全。
 - **Stage 4 (SQL 生成)：**提供 expression（SQL 片段或比率公式），用于生成最终 SQL 查询中的 SELECT 子句。

2. 核心模型配置 (semantic_core.yaml)

- **核心职责：**定义实体（Entities）、维度（Dimensions）、枚举值（Enums）以及实体间的关联关系（Relationships）。
- **消费链路：**
 - **Stage 2 (RAG 检索)：**系统读取维度名称及枚举值构建向量索引，用于召回 DIMENSION_ID 和具体的 Filter Value。
 - **Stage 3 (校验与补全)：**用于校验 Plan 中的维度 ID 是否存在，以及 Filter 中的值是否落在合法的 Enums 范围内。
 - **Stage 4 (SQL 生成)：**提供物理层映射信息，包括物理表名 (table)、物理字段名 (column) 以及表之间的 Join 路径 (relationships)。

3. 安全策略配置 (semantic_security.yaml)

- **核心职责：**定义角色（Roles）、权限范围以及行级安全（RLS）策略片段。
- **消费链路：**
 - **Stage 2 (RAG 检索)：**扮演 [安全门卫] 角色。根据当前用户的 Role 获取权限白名单，在检索阶段直接过滤掉用户无权访问的指标或维度。
 - **Stage 4 (SQL 生成)：**扮演 [执法者] 角色。根据当前用户的 Role，将对应的 RLS 策略片段（如 sales_rep_id = 'u_123'）强制注入到 SQL 的 WHERE 子句中，实现租户隔离与数据权限控制。

4. 通用基础配置 (semantic_common.yaml)

- **核心职责：**定义全局通用的时间窗口模板（如“最近30天”）及通用 NLP 词汇。
- **消费链路：**

- **Stage 3 (校验与补全)**: 用于解析 Plan 中的时间窗口 ID（如 TIME_LAST_30D），将其转换为具体的绝对时间范围（Start Date / End Date），以便后续 SQL 生成。

2. 全局命名空间规范 (Global Naming Conventions)

全局命名空间规范 (Global Naming Conventions)

为了确保语义层在 **LLM 解析**、**后端逻辑推导**、**安全执法** 及 **运维排查** 中的一致性，所有 YAML 配置文件中定义的 **ID (Identifier)** 必须严格遵循以下命名规范。

2.1 核心对象前缀 (Core Objects)

此类对象构成了业务模型的主干，是数据查询的直接载体。

对象类型	必选前缀	示例 (YAML ID)	来源文件	说明
指标 (Metric)	METRIC_	METRIC_GMV, METRIC_AOV	semantic_metrics.yaml	[接口可见] 用户查询的核心目标。
维度 (Dimension)	DIM_	DIM_REGION, DIM_ORDER_DATE	semantic_core.yaml	[接口可见] 用于分组 (Group By) 或筛选 (Filter) 的属性。
实体 (Entity)	ENT_	ENT_SALES_ORDER	semantic_core.yaml	[后端推导] 定义物理表与主键。后端根据 Metric/Dim 自动推导，LLM 不可见。
关系 (Relationship)	REL_	REL_ORDER_TO_USER	semantic_core.yaml	[后端推导] 定义实体间的 Join 路径。

2.2 辅助与逻辑对象前缀 (Auxiliary & Logic Objects)

此类对象用于增强查询能力，提供默认值、枚举约束或复杂的过滤逻辑。

对象类型	必选前缀	示例 (YAML ID)	来源文件	说明
时间窗口 (Time Window)	TIME_	TIME_LAST_30D, TIME_DEFAULT	semantic_common.yaml	[后端兜底] 用于 Stage 3 补全默认时间范围。
逻辑过滤器 (Logical Filter)	LF_	LF_VALID_ORDER, LF_ACTIVE_USER	semantic_core.yaml	[隐式注入] 封装复杂的 WHERE 条件组合，通常绑定在指标上。
状态枚举 (Status Enum)	STATUS_	STATUS_ORDER_FLOW	semantic_core.yaml	代表业务的生命周期或流转节点。通常有先后顺序。必须加前缀。防止与维度名或实体名撞车 例子：订单状态（待支付 -> 已支付 -> 发货 -> 完成）。
类型枚举 (Type Enum)	TYPE_	TYPE_CUSTOMER_LEVEL	semantic_core.yaml	代表事物的分类或属性。通常是并列关系，无先后之分。必须加前缀。防止与维度名或实体名撞车。

2.3 安全与组织对象前缀 (Security & Organization Objects)



此类对象用于定义系统的边界、权限与多租户策略。

对象类型	必选前缀	示例 (YAML ID)	来源文件	说明
角色 (Role)	ROLE_	ROLE_CEO, ROLE_HR_BP	semantic_security.yaml	定义用户角色，用于关联权限策略。
策略 (Policy)	POLICY_	POLICY_CEO_VIEW	semantic_security.yaml	定义角色与数据范围的映射矩阵。
策略片段 (Fragment)	FRAG_	FRAG_SALES_SELF, FRAG_HR_DEPT	semantic_security.yaml	[SQL 注入] 定义具体的 RLS SQL 代码片段（如 id = {{user_id}}）。
行级范围码 (Scope Code)	无 (全大写)	SELF, DEPT, COMPANY	semantic_security.yaml	[系统保留字] 定义权限的逻辑范围，用于连接 Role 与 Fragment。
业务域 (Domain)	无 (全大写)	SALES, HR, FINANCE	semantic_core.yaml	[顶层命名空间] 用于对所有对象进行逻辑分组。



2.4 命名风格约束 (Syntax Constraints)

所有 ID 的字符串格式必须遵守以下 **硬性约束**：

1. 全大写与下划线 (Upper Snake Case)

-  正确： `METRIC_TOTAL_REVENUE`
-  错误： `Metric_Total-Revenue` (驼峰), `metric-total` (中划线)

2. 严禁拼音

-  正确：DIM_CITY
-  错误：DIM_CHENGSHI

3. 字符集限制

- 仅允许使用：A-Z (大写英文字母), 0-9 (数字), _ (下划线)。
- 严禁使用空格、特殊符号或中文字符。

4. 语义清晰

- ID 应具备自解释性，避免使用 METRIC_001 这种无意义的序列号。

5. 全局唯一

- 所有 ID 在全局范围（跨所有 YAML 文件、跨对象类型）必须唯一。
- **注：**ID 指的是包含前缀的完整字符串（如 METRIC_SALES 和 ENT_SALES 是不同的，允许共存；但严禁在两个地方定义完全一样的 METRIC_SALES）。

6. 保留字禁止复用




- **系统保留字：**Scope Code（SELF/DEPT/COMPANY 等）和 Domain 名（如 SALES/HR/FINANCE）视为保留字，其他对象 ID 严禁使用这些字面值。
- **技术保留字：**严禁与 SQL 关键字（如 SELECT, FROM, WHERE, NULL, ALL）同名，以防下游 SQL 生成或日志排查踩坑。

3. 结构概述

3. 配置模型详解 (Schema Specifications)

本章对 4 个核心 YAML 文件的内部结构进行字段级定义。

图例说明：

- **必填：** (必须存在，不可为 null)
- **选填：** (可以省略，代码需处理默认值或 None)
- **条件：** (依赖其他字段的值)

3.1 核心模型 (semantic_core.yaml)

此文件定义了数据仓库的**物理骨架与逻辑实体**。它是 Stage 4 生成 SQL 的基础。

3.1.1 顶层结构概览

代码块

1

domains: [...]

业务域定义

2

entities: [...]

实体定义 (物理表映射)

3

dimensions: [...]

维度定义 (字段映射)

4

enums: [...]

枚举值集合

5

placeholders: {...}

预留扩展槽

6

logical_filters: [...]

逻辑过滤器 (复杂 WHERE 条件封装)

7

relationships: [...]

实体关联关系 (Join 路径)

3.1.2 字段详细定义

1. 业务域 (Domains)

用于对指标和实体进行顶层逻辑分组。

字段名	类型	必填	说明	示例
id	String	<input checked="" type="checkbox"/>	业务域唯一标识。全大写。	SALES, HR
name	String	<input checked="" type="checkbox"/>	中文显示名称。	"销售域"
default_entity_id	String	<input type="checkbox"/>	该域下的默认主实体 ID。用于 Stage 3 兜底推导。	ENT_SALES_ORDER_ITEM

2. 实体 (Entities)

定义业务对象与物理表（或视图）的映射关系。

字段名	类型	必填	说明	示例
id	String	<input checked="" type="checkbox"/>	实体唯一标识。 必须以 ENT_ 开头。	ENT_EMPLOYEE
name	String	<input checked="" type="checkbox"/>	实体中文名称。	"员工"
aliases	List[Str]	<input type="checkbox"/>	实体的自然语言别名，用于 RAG 检索。	["职员", "同事"]
domain_id	String	<input checked="" type="checkbox"/>	所属业务域 ID。	HR
entity_type	Enum	<input checked="" type="checkbox"/>	FACT (事实表) 或 DIMENSION (维度表)。	FACT
semantic_view	String	<input checked="" type="checkbox"/>	物理层映射。 对应数据库中的 View 或 Table 名。	v_sales_order_item
primary_key	List[Str]	<input checked="" type="checkbox"/>	物理主键字段列表。用于去重统计。	["order_id"]
default_time_field_id	String	<input type="checkbox"/>	默认时间维度 ID。用于 Stage 3 补全 TREND 查询。	ORDER_DATE

3. 维度 (Dimensions)

定义可用于 GROUP BY 或 WHERE 的属性。

字段名	类型	必填	说明	示例
id	String	<input checked="" type="checkbox"/>	维度唯一标识。 必须以 DIM_ 开头 。	DIM_CITY
name	String	<input checked="" type="checkbox"/>	维度中文名称。	"城市"
aliases	List[Str]	<input type="checkbox"/>	维度别名，用于 RAG 检索。	["地区", "所在城市"]
entity_id	String	<input checked="" type="checkbox"/>	所属实体 ID。表明该维度属于哪张表。	ENT_SALES_ORDER
column	String	<input checked="" type="checkbox"/>	物理层映射 。对应数据库表中的物理列名。	city_name
data_type	Enum	<input checked="" type="checkbox"/>	STRING, INTEGER, DATE, BOOLEAN。	STRING
is_time_dimension	Bool	<input type="checkbox"/>	是否为时间维度。默认为 false。	TRUE
time_field_id	String	<input type="checkbox"/>	若 is_time_dimension=true，此字段必填。定义时间语义类型。	ORDER_DATE
allowed_time_grains	List	<input type="checkbox"/>	若为时间维度，定义支持的粒度。	["DAY", "MONTH"]
enum_value_set_id	String	<input type="checkbox"/>	关联的枚举集合 ID。用于限定 Filter 取值范围。	STATUS_VALID_FOR_REV

4. 枚举值集合 (Enums)

定义维度的合法取值范围（Value Set），用于 RAG 检索和 Filter 校验。

字段名	类型	必填	说明	示例
id	String	<input checked="" type="checkbox"/>	枚举集合 ID。 必须以 STATUS_ 或 TYPE_ 开头 。	STATUS_ORDER_ALL
name	String	<input checked="" type="checkbox"/>	集合名称。	"订单状态全集"
target_dim_id	String	<input checked="" type="checkbox"/>	绑定的维度 ID。	DIM_ORDER_STATUS
values	List[Str]	<input checked="" type="checkbox"/>	合法的枚举值列表（字符串字面量）。	["Shipped", "Paid"]

5. 逻辑过滤器 (Logical Filters)

封装复杂的 SQL 过滤逻辑，供指标引用。

字段名	类型	必填	说明	示例
id	String	<input checked="" type="checkbox"/>	过滤器 ID。 必须以 LF_ 开头 。	LF_VALID_ORDER
filters	List[Obj]	<input checked="" type="checkbox"/>	包含一组具体的过滤条件。	(见下表结构)

⚠ 约束 (Constraint):

- **定位：** 本配置（Logical Filters）仅作为 **后端宏定义 (Macro Definition)**。
- **用途：** 仅供 `semantic_metrics.yaml` 中的 `default_filters` 字段引用，用于简化重复的过滤逻辑配置。
- **禁区：严禁** 将此列表注入 Stage 2 的 Schema Context。LLM 不需要感知逻辑过滤器的存在，它只能看到物理维度和枚举值。

filters 列表元素结构：

- **常规模式：** { "target_id": "DIM_ID", "operator": "IN_SET", "value_set_id": "ENUM_ID" }
- **Raw SQL 模式：** { "operator": "RAW_SQL", "sql": "status != 'deleted'" }

6. 关系 (Relationships)

定义实体间的连接路径（Join Path）。

字段名	类型	必填	说明	示例
id	String	✓	关系 ID。必须以 REL_ 开头	REL_ORDER_USER
left_entity	String	✓	左表实体 ID。	ENT_SALES_ORDER
right_entity	String	✓	右表实体 ID。	ENT_USER
join_type	Enum	✓	LEFT, INNER。MVP 推荐默认 LEFT。	LEFT
conditions	List	✓	连接键对。[{ "left": "user_id", "right": "id" }]	-

⚠ **V1 阶段限制：** 本配置（Relationships）仅用于 RLS 权限推导或元数据展示。**Stage 4 SQL 生成器** 严禁使用此配置生成 JOIN 子句（所有查询必须在 Semantic View 宽表内闭环）。

3.2 指标体系 (semantic_metrics.yaml)

此文件定义了业务的核心查询对象。它是 RAG 检索的重点。

3.2.1 顶层结构概览

代码块

```
1 metrics: [...] # 指标列表
```

3.2.2 Metrics 字段详细定义

字段名	类型	必填	说明	示例
id	String	<input checked="" type="checkbox"/>	指标唯一标识。 必须以 METRIC_ 开头。	METRIC_GMV
name	String	<input checked="" type="checkbox"/>	指标标准名称。	"销售额"
aliases	List[Str]	<input checked="" type="checkbox"/>	关键。 指标同义词，用于 RAG 召回。	["营收", "GMV"]
description	String	<input type="checkbox"/>	业务口径描述，辅助 LLM 理解。	"不含退款的订单金额"
domain_id	String	<input checked="" type="checkbox"/>	所属业务域。	SALES
entity_id	String	<input checked="" type="checkbox"/>	主表绑定。 指标所属的实体 ID。	ENT_SALES_ORDER_ITEM
metric_type	Enum	<input checked="" type="checkbox"/>	AGG (基础聚合) 或 RATIO (衍生比率)。	AGG
expression	Object	<input checked="" type="checkbox"/>	计算公式定义 (见下文)。	{ "sql": "SUM(amt)" }
default_time	Object	<input type="checkbox"/>	默认时间窗口配置 (见下文)。	(见下文)
default_filters	List[Str]	<input type="checkbox"/>	强制绑定的逻辑过滤器 ID 列表 (LF_)。	["LF_VALID_ORDER"]
unit	Enum	<input type="checkbox"/>	单位类型。CURRENCY, COUNT, PERCENT。	CURRENCY
is_additive	Bool	<input checked="" type="checkbox"/>	是否可加。比率指标通常为 false。	TRUE

附：expression 结构详解

场景 A：基础聚合 (metric_type="AGG")

代码块

```
1  expression:
2    expr_type: "SQL"
3    sql: "SUM(line_amount)"  # 物理字段的聚合表达式
```

场景 B：比率指标 (metric_type="RATIO")

代码块

```
1  expression:
2    expr_type: "METRIC_RATIO"
3    numerator_metric_id: "METRIC_GMV"      # 分子指标 ID
4    denominator_metric_id: "METRIC_UV"     # 分母指标 ID
5    safe_division: true                    # 是否处理除零异常
```

⚠ **实体一致性约束：** 分子指标 (`numerator`) 和 分母指标 (`denominator`) 必须归属于 **同一个 Entity (即同一个 Semantic View)**。V1 版本不支持跨实体 (Cross-Entity) 的比率计算。

附：default_time 结构详解

用于 Stage 3 在用户未指定时间时进行兜底。

代码块

```
1  default_time:
2    time_field_id: "ORDER_DATE"           # 默认使用哪个时间字段
3    time_window_id: "TIME_LAST_30D"      # 引用 common 中的窗口模板
```

3.3 安全策略 (semantic_security.yaml)

此文件定义了数据访问的控制平面。它是 Stage 4 生成 SQL 时的“执法依据”。

3.3.1 顶层结构概览

代码块

```
1  roles: [...]           # 角色定义
2  row_scopes: [...]      # 范围类型定义 (SELF/DEPT/COMPANY)
3  policy_fragments: [...] # RLS SQL 片段库
4  row_scope_bindings: [...] # 范围与片段的绑定关系
5  role_policies: [...]   # 角色与范围的最终矩阵
```

3.3.2 字段详细定义

1. 角色 (Roles)

字段名	类型	必填	说明	示例
role_id	String	<input checked="" type="checkbox"/>	角色 ID。必须以 ROLE_ 开头 。	ROLE_SALES_MANAGER
name	String	<input checked="" type="checkbox"/>	角色名称。	"销售经理"

2. 策略片段 (Policy Fragments)

定义具体的 SQL 注入逻辑。

字段名	类型	必填	说明	示例
fragment_id	String	✓	片段 ID。必须以 FRAG_ 开头。	FRAG_SALES_SELF
domain_id	String	✓	作用的业务域。	SALES
entity_id	String	✓	作用的实体表。	ENT_SALES_ORDER
raw_condition	String	✓	SQL 模板 。支持 Jinja2 语法注入上下文变量。	sales_rep_id = {{user_id}}

3. 范围绑定 (Row Scope Bindings)

将抽象的范围代码映射到具体的 SQL 片段。

字段名	类型	必填	说明	示例
row_scope_code	String	✓	范围代码。SELF, DEPT, COMPANY。	SELF
bindings	List	✓	绑定列表。	(见下)

Binding 元素结构: { "domain_id": "SALES", "entity_id": "ENT_ORDER", "fragment_ref": "FRAG_SALES_SELF" }

4. 角色策略矩阵 (Role Policies)

定义每个角色能看什么。

字段名	类型	必填	说明	示例
policy_id	String	✓	策略 ID。必须以 POLICY_ 开头。	POLICY_SALES_MGR
role_id	String	✓	绑定的角色 ID。	ROLE_SALES_MANAGER
scopes	Object	✓	权限范围配置对象。	(见下)

Scopes 对象结构:

- domain_access: ["SALES", "HR"] (允许访问的域)
- row_scope_code: DEPT (行级权限范围，引用 row_scopes)
- metric_scope: ["ALL"] (指标白名单，ALL 或具体 ID 列表)

3.4 通用配置 (semantic_common.yaml)

此文件定义全局共享的资源。

3.4.1 顶层结构概览

代码块

```
1  global_settings: {...}    # [新增] 全局默认配置
2  time_windows: [...]      # 时间窗口模板
3  common_vocabulary: [...] # 通用词典
```

3.4.2 字段详细定义

1. 全局设置 (Global Settings)

字段名	类型	必填	说明	示例
default_time_window_id	String	<div>✓</div>	全局默认的时间窗口 ID。必须引用 time_windows 列表中已定义的 id。	TIME_LAST_30D

2. 时间窗口 (Time Windows)

字段名	类型	必填	说明	示例
id	String	<div>✓</div>	窗口 ID。必须以 TIME_ 开头。	TIME_LAST_7D
template	Object	<div>✓</div>	窗口计算逻辑。	{ "type": "LAST_N", "value": 7, "unit": "DAY" }

3. 通用词汇 (Common Vocabulary)

用于增强 Stage 2 对非实体类词汇的理解。

字段名	类型	必填	说明	示例
term	String	<div>✓</div>	标准词。	"同比"
aliases	List[Str]	<div>✓</div>	同义词列表。	["YoY", "比去年"]
value	String	<div>✓</div>	对应的系统枚举值。	YOY

4. 维护与变更指南 (Lifecycle & Maintenance)

本章定义了语义层配置在全生命周期中的变更流程。为了确保线上服务的稳定性，所有配置变更必须严格遵循“**修改 -> 校验 -> 发布 -> 生效**”的标准作业程序。

4.1 场景一：新增或修改指标 (New/Modify Metrics)

场景描述：业务方提出新的分析需求（如“新增‘毛利率’指标”）或原有指标口径发生变更。

- **涉及文件：** semantic_metrics.yaml
- **操作步骤：**
 - a. **定义 ID：**遵循 METRIC_ 前缀规范，确保 ID 全局唯一。
 - b. **编写逻辑：**
 - 若为基础指标，编写 SQL expression。
 - 若为衍生指标，编写 Ratio 分子分母依赖。
 - c. **配置别名：**在 aliases 中尽可能多地添加业务术语（如“毛利”、“Gross Margin”），以提高 RAG 召回率。
 - d. **绑定实体：**明确指定 entity_id（如 ENT_SALES_ORDER）。
- **生效机制：**
 - **[必须重建索引]：**修改指标定义后，必须 **重启服务** 或 **调用 RAG 索引重建接口**。
 - **原因：**指标的元数据（名称、别名、描述）需要被 Embedding 模型转化为向量并写入向量数据库。仅修改文件而不重建索引，LLM 将无法感知新指标。

4.2 场景二：添加同义词 (Add Synonyms / Tuning)

场景描述：用户反馈某些“黑话”或缩写无法识别（如用户搜“GMV”能查到，搜“流水”查不到）。

- **涉及文件：**
 - 若是特定指标的别名：修改 semantic_metrics.yaml 中的 aliases。
 - 若是特定维度的别名：修改 semantic_core.yaml 中的 aliases。
 - 若是通用修饰词（如“同比”）：修改 semantic_common.yaml 中的 common_vocabulary。
- **操作步骤：**
 - a. 找到对应对象的定义节点。
 - b. 在 aliases 列表中追加新的词汇字符串。
- **生效机制：**
 - **[必须重建索引]：**必须 **重启服务** 或 **调用 RAG 索引重建接口**。
 - **原因：**同义词是向量检索的核心依据，必须更新向量库才能生效。

4.3 场景三：调整权限规则 (Adjust Permissions)

场景描述：组织架构调整，或者新入职员工需要分配角色。

- **涉及文件：** semantic_security.yaml
- **操作步骤：**
 - a. **新增角色：**在 roles 列表中定义新角色 ID（如 ROLE_REGIONAL_MGR）。
 - b. **定义策略：**在 role_policies 中配置该角色对 Domain、Metric、Row Scope 的访问权限。
 - c. **绑定 RLS：**如有特殊的行级过滤需求，需在 policy_fragments 中编写新的 SQL 片段。
- **生效机制：**
 - **[无需重建索引]：**权限配置（角色、策略）不参与 RAG 向量检索，因此不需要重新 Embedding。
 - **[仅需刷新缓存]：**只需 **热重载配置 (Hot Reload)** 或 **重启应用**，让系统重新加载内存中的权限矩阵即可生效。

4.4 场景四：数据库表结构变更 (DB Schema Changes)

场景描述：底层数据仓库（DWD/DWS）发生了 DDL 变更（如字段改名、新增字段）。

- **涉及文件：** semantic_core.yaml
- **操作步骤：**
 - a. **定位实体：**找到对应的 entities 定义。
 - b. **修改映射：**
 - 若表名变更：更新 semantic_view。
 - 若字段变更：找到对应的 dimensions 节点，更新 column 字段。
 - c. **验证：**务必确保 YAML 中的 column 与数据库中的实际列名 **100% 一致**。
- **生效机制：**
 - **[刷新配置]：****重启应用** 以加载最新的物理映射配置。
 - **[高危风险提示]：**此类变更必须与数据库 DDL 变更保持 **原子性发布**。若 YAML 改了但数据库没改（或反之），Stage 5 执行 SQL 时会直接报错（Column not found）。

变更影响速查表 (Impact Matrix)

变更场景	修改文件	影响阶段	是否需要重建向量索引?	风险等级
改指标口径	metrics.yaml	Stage 4 (SQL)	否 (仅改 SQL 逻辑)	中
加指标/维度	metrics/ core.yaml	Stage 2 (RAG)	是 (必须)	低
加同义词	metrics/ core.yaml	Stage 2 (RAG)	是 (必须)	低
改权限	security.yaml	Stage 2/4	否 (仅刷内存)	高 (可能泄露数据)
改物理表名	core.yaml	Stage 4/5	否 (仅刷内存)	极高 (服务中断)

附录

附录A 设计理念的细节思考

1. 为什么这套语义层的设计满足项目需求？

本规范采用的“四文件分层结构”是基于以下三个工程目标的平衡结果：

1. 开发效率 (Engineering Efficiency)

- 结构清晰且强类型化。Cursor/Copilot 等 AI 编程工具能极快地理解 Schema，并自动生成对应的 Pydantic 模型代码，显著降低了后端解析层的开发成本。

2. RAG 检索效果 (RAG Performance)

- 遵循“高内聚 (High Cohesion)”原则。我们将指标的别名 (Aliases) 与定义放在一起，将维度的枚举值 (Enums) 与维度定义放在一起。
- 收益：构建向量索引时，单个 Chunk 包含了完整的语义上下文，解决了“搜索别名找不到正主”的召回难题。

3. 安全解耦 (Security Decoupling)

- 独立的 semantic_security.yaml 确保了安全逻辑与业务逻辑的物理隔离。
- 收益：防止业务人员在修改指标口径时，意外破坏了底层的权限控制规则。

2. 为什么enum要放在core中？而不单独放到一个yaml中？

关于“枚举值是否应该放在 semantic_core.yaml 中”，我们经历了深入的权衡，最终制定了“二分法策略”。

1. 权衡分析 (Trade-off Analysis)

- 放在 Core 中的优势（利）：
 - 逻辑强绑定：维度（如“订单状态”）与取值集合（如“已发货/已完成”）在业务上是不可分割的。
 - RAG 友好：LLM 在检索到维度时，能立即感知其合法取值，从而生成准确的 Filter 条件。
- 放在 Core 中的风险（弊）：
 - 主数据爆炸：如果将“国家列表（200+）”或“SKU 列表（10000+）”放入 YAML，会导致文件体积失控，且挤占 LLM 的 Context Window。

2. 最终策略：枚举分层 (Stratification Strategy)

为了解决上述矛盾，我们严格将枚举分为 A 类和 B 类，分别处理：

分类	A 类：业务逻辑枚举 (Business Logic Enums)	B 类：主数据字典 (Master Data)
定义	描述业务流程状态或逻辑分组的有限集合。	描述具体业务对象的实体清单，通常数量巨大且动态变化。
特征	数量少 (< 20)，相对静态。	数量多 (> 50)，经常变动。
示例	订单状态、支付方式、员工在职状态。	客户名单、SKU 列表、门店名称、城市列表。
处理方式	写入 YAML (semantic_core.yaml)	严禁写入 YAML
理由	这是 LLM 理解业务逻辑的关键，必须显式透出给模型。	LLM 不需要背诵所有客户名字也能写出 SQL。

3. B 类主数据的处理原则

对于 B 类数据（如“客户名称”），系统遵循“盲填 + 后端校验”的模式：

- LLM 行为：
 - 用户问：“客户张三的订单...”
 - LLM 不需要知道“张三”是否在 YAML 里。它只需识别出这是 DIM_CUSTOMER_NAME，并直接输出 Filter: {id: "DIM_CUSTOMER_NAME", values: ["张三"]}。
- 后端行为：
 - Stage 4 生成 SQL 时，直接将“张三”填入 SQL 的 WHERE 子句。
 - 校验责任：由底层数据库（Database）或独立的主数据服务负责校验“张三”是否存在。如果不存在，SQL 执行结果为空，符合预期。

总结：

YAML 仅承载 **驱动业务逻辑流转的元数据 (Metadata)**，即定义了业务规则边界的有限集合（如状态、类型）；而 **海量的具体业务对象 (Instance Data)**，即数量巨大且动态变化的实体记录（如客户名称、商品编码），应交由数据库存储与检索，严禁硬编码在配置中。

3. 为什么每一个 Metric 和 Dimension 都强制要求配置 entity_id?

一句话答案：为了把“不确定的预测”变成“确定的查表”。

当初我们设计 YAML 结构时，面临两个选择：

- **方案 A (传统做法)：让 LLM 预测实体**
 - Prompt：用户问的是销售额，请判断属于哪个表？
 - LLM 输出：Table: sales_orders
 - **风险**：LLM 可能会幻觉，可能会拼错表名，或者在“订单主表”和“订单明细表”之间摇摆不定。
- **方案 B (我们的做法)：配置即真理 (Configuration as Truth)**
 - **逻辑**：在业务定义中，“销售额 (Metric)”天然就生长在“销售订单行 (Entity)”上。这是一种物理属性，就像“苹果长在苹果树上”一样，是不变的。
 - **操作**：我们在 semantic_metrics.yaml 里写死 METRIC_GMV -> entity_id: SALES_ORDER_ITEM。
 - **收益**：
 - LLM 减负**：LLM 只需要听懂“销售额”这三个字（识别 Metric ID），不需要懂数据库结构。
 - 100% 准确**：只要 Metric ID 对了，Entity ID 绝对错不了，因为是查字典查出来的。
 - 解耦**：未来如果你把底层表从 MySQL 换成了 ClickHouse，或者改了表名，只需要改 YAML 配置，不需要重新微调 LLM。

这就是为什么我们要强制配置 entity_id：用后端配置的确定性，去对冲 LLM 的不确定性

附录B 设计缺陷，以及未来演进方向

本附录记录了语义层设计中的关键权衡 (Trade-offs)，特别是关于“配置维护成本”与“系统灵活性”之间的博弈。

B.1 为什么 MVP 阶段坚持使用 YAML? (The "Config-as-Code" Decision)

在 V1 版本中，我们选择 **YAML 文件** 作为语义层的唯一载体，而非开发图形化管理后台 (Admin UI)，是基于以下考量：

1. **版本控制 (GitOps)**: 指标口径是企业的核心资产。YAML 配合 Git, 天然具备变更审计、版本回滚和 Code Review 能力, 这是数据库存储难以低成本实现的。
2. **表达能力**: 复杂的 SQL 逻辑 (如嵌套 Case When) 在 YAML 中极易表达, 而在低代码 UI 中开发成本极高。
3. **交付速度**: 避免了开发复杂的 “指标管理后台”, 将资源聚焦于核心的 NL2SQL 链路。

B.2 演进路线图: 如何解决 “维护累” 的问题?

我们承认, 随着业务发展, 手动修改 YAML 会面临 “操作繁琐” 和 “门槛过高” 的问题。系统将按以下阶段演进:

- **阶段一: 人工配置 (Current MVP)**
 - **模式**: 后端/数据工程师手动修改 YAML -> 发布上线。
 - **适用**: 指标变更频率较低, 团队规模较小。
- **阶段二: 低代码后台 (Admin UI)**
 - **模式**: 业务人员在 Web 界面填表 -> 后端自动生成/修改 YAML 并推送到 Git。
 - **本质**: UI 是壳, YAML 是核。保持了 GitOps 的优势, 降低了操作门槛。
- **阶段三: Headless BI 集成**
 - **模式**: NL2SQL 系统不再维护自己的语义层, 而是通过 API 实时同步公司现有指标平台 (如 Kyligence, QuickBI) 的元数据。

B.3 代码层面的预留设计 (Future-Proofing)

为了支持上述演进, 我们在代码架构上采用了 “接口隔离模式 (Interface Segregation)”。

设计方案简述:

核心组件 SemanticRegistry 不直接依赖文件系统, 而是依赖抽象的 ConfigurationProvider 接口。

代码示例 (Python):

代码块

```
1  from abc import ABC, abstractmethod
2
3  # 1. 定义抽象接口 (Interface)
4  class SemanticConfigProvider(ABC):
5      """
6      配置提供者基类。
7      未来无论是读 YAML、读数据库、还是调 API, 都实现这个接口。
8      """
9      @abstractmethod
10     def load_metrics(self) -> list[MetricDef]:
11         pass
```

```

12
13     @abstractmethod
14     def load_security_policies(self) -> list[PolicyDef]:
15         pass
16
17     # 2. MVP 实现: 基于 YAML (Current Implementation)
18     class YamlFileProvider(SemanticConfigProvider):
19         def load_metrics(self):
20             # 读取本地 semantic_metrics.yaml
21             return yaml.safe_load(open("semantics/semantic_metrics.yaml"))
22
23     # 3. 未来实现: 基于数据库/低代码平台 (Future Implementation)
24     class DatabaseProvider(SemanticConfigProvider):
25         def load_metrics(self):
26             # 从配置表读取, 支持热更新
27             return db.query("SELECT * FROM sys_metric_definitions")
28
29     # 4. 启动时注入
30     # 只需要改一行代码, 就能从 YAML 模式切换到 DB 模式
31     # provider = YamlFileProvider()
32     provider = DatabaseProvider() # Future switch
33     registry = SemanticRegistry(provider)

```

结论:

当前的“笨办法”是为了未来的“大智慧”留出空间。只要接口定义得当，底层的存储介质切换对上层业务逻辑是透明的。