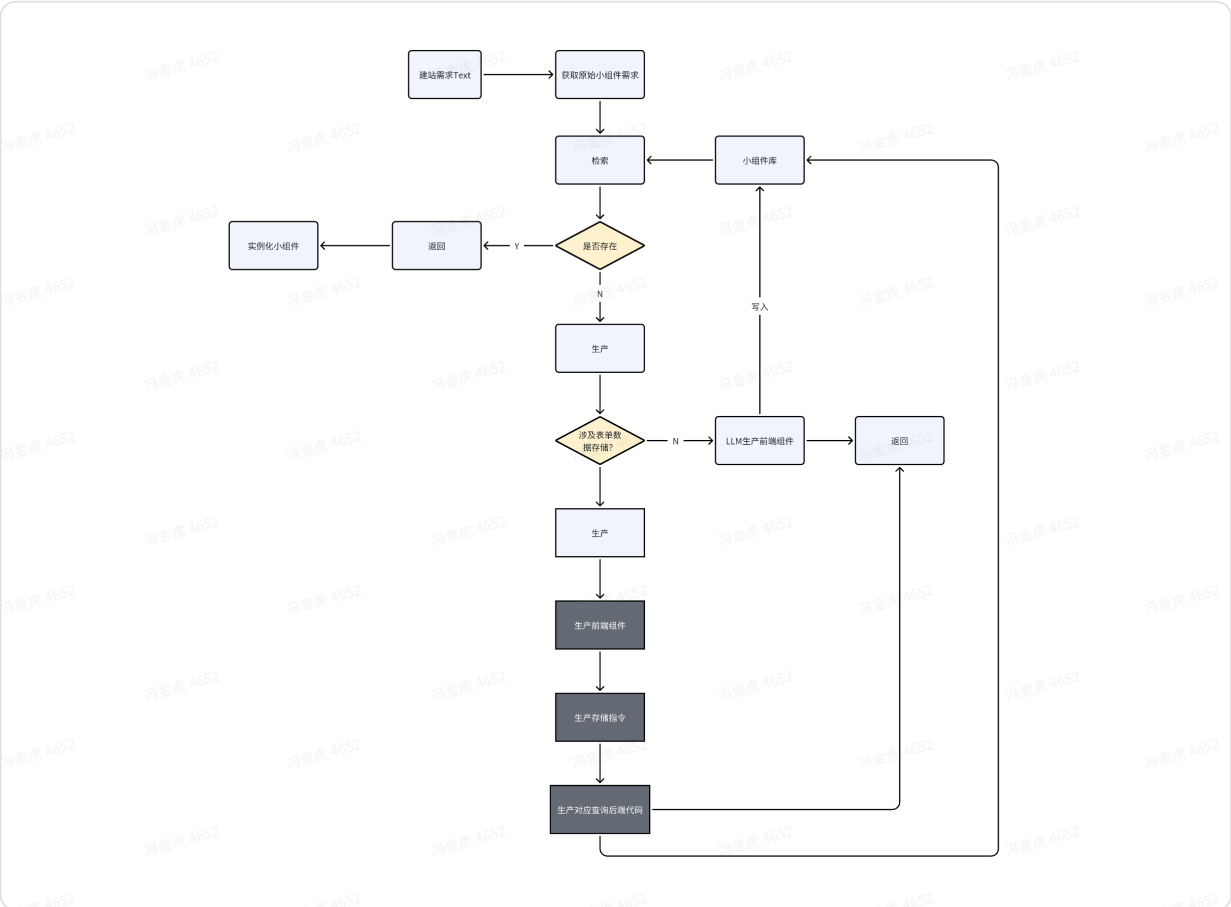
小组件灵活存储的后端系统设计

背景

为丰富生成网站和局部修改的丰富度和能力，设计了小组件功能，这里要设计一套系统，来存储各种各样即时生产的小组件产生的各式异构数据。同时，要考虑这些存储的查询，包括写接口和读接口设计。

业务流程



从上述一般的业务流程来看，不涉及后端存储的小组件这里不用care, 剩下的只关注涉及后端存储的小组件，也即上述黑色的3个流程框。

无论是预先生成还是实时生成的组件，都会被缓存，然后后续在小组件库中被检索。

这里生产的小组件还分3种情况。

1. 预先设计生成
2. 实时LLM生成
3. 提供了较为通用的后端接口，供开发者使用来生产具有数据存储能力的小组件

这3种情况对应的解决方案，最好的做法当然是一种通用手段能覆盖3种需求，但是实际上我们可能很难找出这种方案，甚至这种方案不一定是综合来看最划算的，尤其是考虑到他们的优先级以及实施的工期问题。下面分别讨论3种情况的解决方案，综合共性以及优劣，给出最终的选型建议参考。

总体思路：

第一步：各种方案是怎么回事讲清楚

第二步：总结每个方案的特点（从角色和可做的操作（生成、更改）两方面，组件类型和组件实例是不同的）安全性、可操作性、生成效率、数据的操作（存、查）、指出生成阶段、更新阶段等。

第三步：列表格直观比较

第四步：选型决策

选型及设计论述

主要涉及两方面，一是后端存储，一是后端业务逻辑

这种模式的特点是，可以预先设计异构的数据如何处理，以及接口如何处理，写好代码，被选中的小组件直接调用即可。现在我们基于这个最简单的方式来设计方案。

这里主要面临两个任务，一是如何设计存储Schema，来存储各种异构的数据，同时保障一定的查询效率；二是提供对这些异构数据的操作接口——路径、参数及返回值设计。

传统的页面数据存储一般是提出功能->抽象数据->设计表结构->按功能写查询接口->优化索引->考虑数据规模分库分表等等。基本是一个功能或者数据结构对应一套从数据设计到查询设计的完整链路，由研发人工保障整体的功能、效率、安全性。

后来出现一种低代码的平台设计场景，即预设一些基本组件，组件可自由可视化拖拽组合，灵活实现一定的业务场景。这里的组合，对应于前端的组件JSON组合，也对应于后端的接口参数更改、数据结构灵活变更、查询内容的变更。

以上描述来看，我们的场景和上述有一定的相似性，（但是实际上形似而神不似，差别还是挺大，低代码平台很多固化的场景和流程， 使用的协议和设计可能特别复杂，我们没有这个时间和团队来支持）除了不太有组件组合场景外，其他的诸如业务场景自由度比较高，异构数据较多，而且要实现从查询接口到数据最终落地的一整个链路等特性比较类似。

这里列出几种常见的解决思路。

传统开发法

即每个相同结构类型对应一个表和处理接口。行模型是数据库设计的标准数据建模技术。

因为组件也是一个个开发的，每个小组件对应一个不同的业务逻辑。所以每开发一个组件，即开发一套对应的后端组件逻辑元组（接口路径-接口参数-流程-数据结构(数据表)-数据库）。**即同一类型的小组件使用一个后端逻辑元组**

这种方法就和低代码完全不同，因为低代码的定义就限制了它是可视化编辑，高度复用，而这个方法就是传统的代码编辑与开发。如果有复用，也只是相似的业务组件可以小范围人工抽象，其性价比似

乎还比不上一个组件一个开发元组。

直接小组件ID加页面ID查询，会完全泄露隐私（需要有相应的鉴权手段）。

优点:

开发逻辑简单，流程明晰，高度支持小组件的几乎所有功能

缺点：

用户端几乎无法修改（除非做预留设计），复用度很低，开发量大，不灵活，每个小组件都需要对应的后端提供开发。

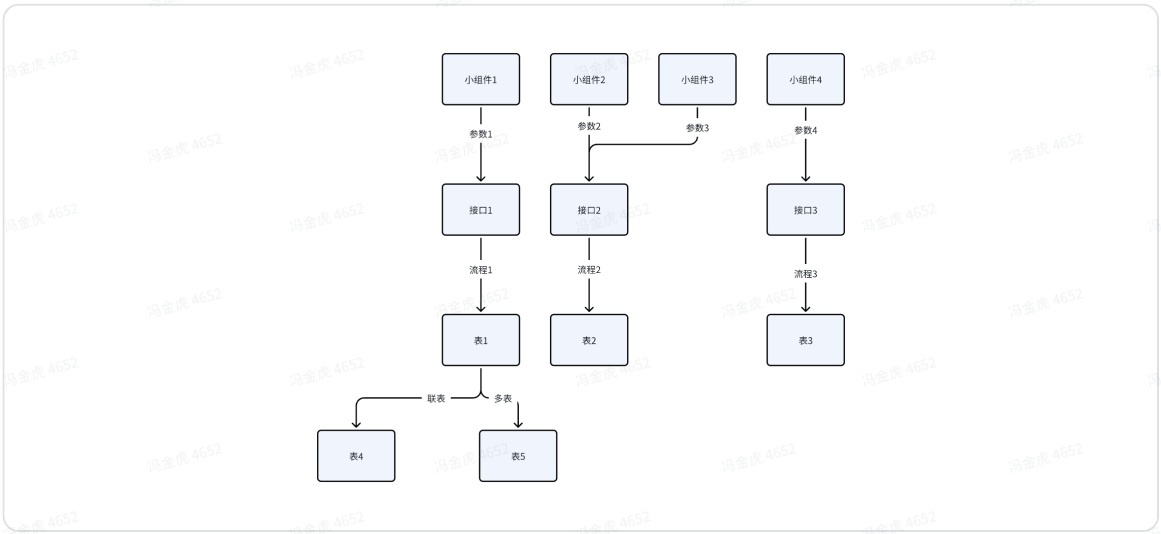
特点总结：

前端：按类型生产小组件

新建：研发手动设计

修改：研发手动设计和修改

其他人：无法修改



使用单文件

客户的某个页面的小组件（唯一ID）对应一个文件，类似CSV的文件，当然也可以做成大Json. 每次读取都是读整个Json, 传给前端，所有查询过滤等逻辑均做到前端。后端只需要接整个数据整个更新或者整个返回即可。更新时需加分布式锁，所有针对单个文件的写操作都需串行。**一个组件实例使用一个文件，而不管类型是否相同**

如果要加更复杂筛选功能，比如权限控制部分数据行或者列的返回，后端的工作则大增，资源消耗尤其是内存消耗会飙升。

优点：

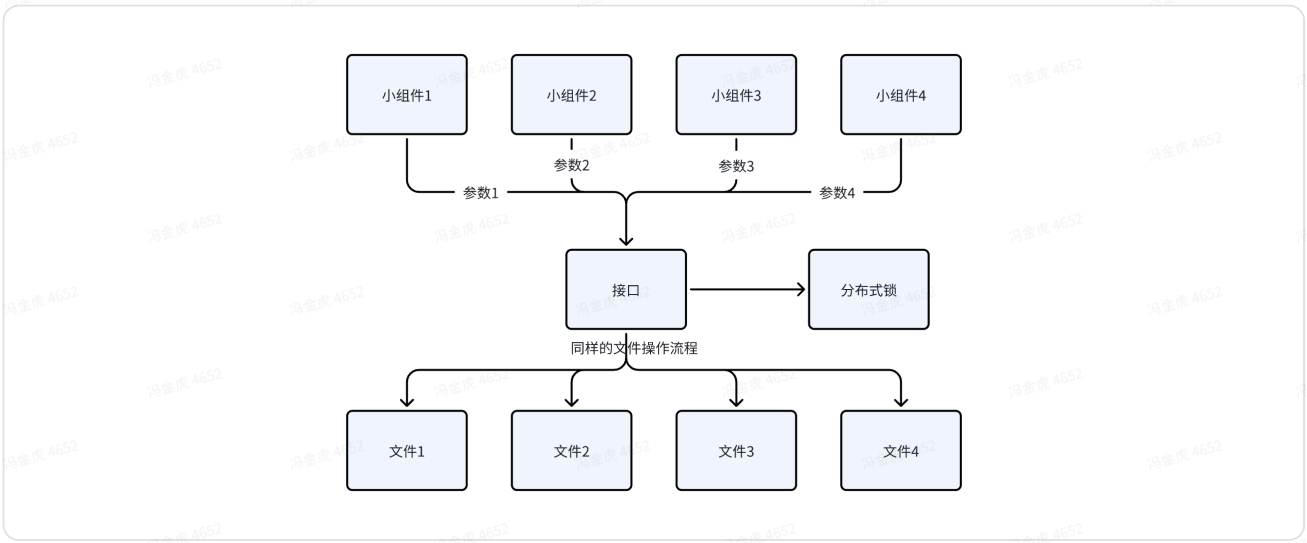
接口完全复用，实现简单，贴近前端，所有数据操作逻辑均由前端实现；用户端可以直接修改结构（需前端处理历史数据）；数据天然隔离。

缺点：

无法做权限校验等行级别逻辑操作，性能低下，存在一定安全风险

无法做多表联查等对应的数据库操作，这样一来只能支持平铺的单实体逻辑

难点：客户端存在并发的情况下，不能使用数据快照来做更新数据操作，而是使用后端语义的append/delete等做单条处理，后端加分布式读写锁来处理数据语义



横向设计法

程序生成动态实体，行模型是数据库设计的标准数据建模技术

即每新建一个组件实例，则新建一个表（或者多个表，即表组）（把单文件直接用表实现），不用管复用的事。

那么组件实例对应的结构更新对应着表的schema更新，新建组件需要提交表单数据的Schema，更新表单结构需要给出Modify，对于数据库总之是DDL操作。

这里理论上是可以做联表、多表查询等复杂操作的，毕竟是数据库实现的，但是无法做自动化或者自动化的代价太大。

EAV（行代替列）

E即实体，也即我们需要存储的对象，这里要支持的对象是各种各样的；A即属性，各种不同对象的不同属性，同一对象的属性也会变更；V即某个对象的某个属性的真实值。

实体表E

实体ID	实体类型	实体名称
1	Product	商品1号
2	Post	帖子2号

属性表A

属性ID	属性名称	值类型
1	评论内容	string
2	更新时间	date
3	价格	int

值表V（优化方式： v-{type}）

实体ID	属性ID	值
2	1	hello
2	2	2014-08-29
1	3	15

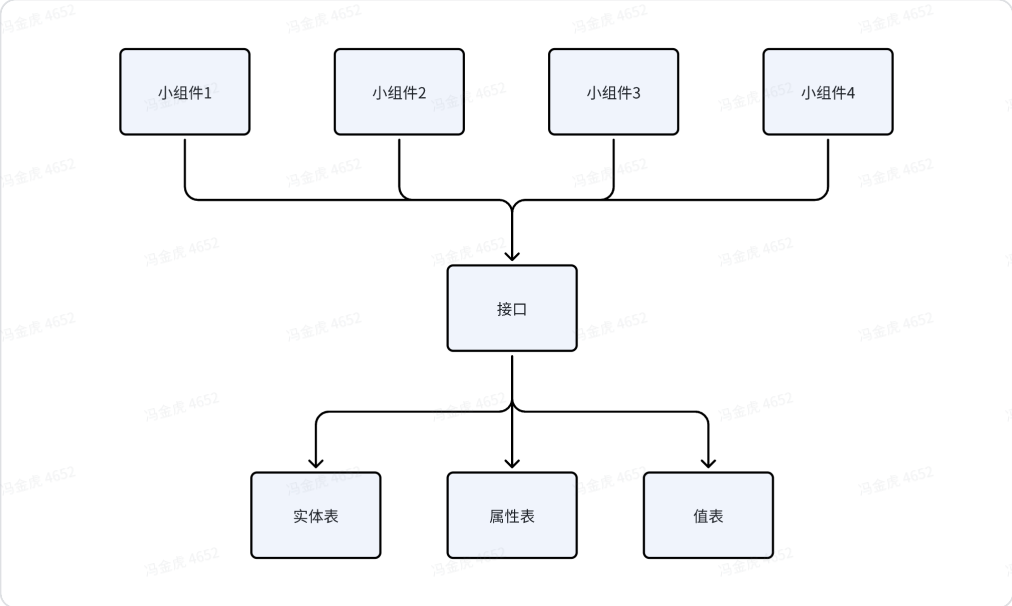
优点：

- **灵活性高**：能够轻松添加或删除属性，无需修改表结构。
- **扩展性强**：支持无限多的属性，满足复杂数据管理的需求。
- **节省空间**：避免了传统数据库中的宽表问题，减少了存储空间浪费。

缺点：

- **查询复杂**：由于数据分布在多个表中，查询时需要进行多次表连接，可能影响性能。
- **数据完整性维护困难**：在更新或删除数据时，需要确保数据的一致性和完整性。
- **性能问题**：在数据量较大或者列较多时，查询性能低下。

在我们这个场景下的使用方式：



新建小组件：

表都是建好的，后端不用做任何动作

更新小组件：

表还是建好的，后端不用做任何动作

某个小组件提交了新建数据（一个Json，包含数据本体和一些固定meta），后端需要固定流程处理，先新建实体，然后查找到属性或者新建属性，然后组成值表的插入行数组，执行值表写入

所有属性都是跟随具体的实体记录变动的，所以也无所谓结构的变更了，每个实体本身在值表里就对应了一组具体的属性，已经是完全的Schema-less了。

查询需要3表联动查询，这里就不展开了。

这里还需要做的是实体表里要加上Type作为不同类型（以前的Table，比如是一个具体的小组件）的载体。

安全和隔离层面看，每个客户或者每个网站可以使用一套EAV表3元组。

元信息+宽表

元信息 (Metadata)：元信息存储与实体、属性相关的配置、定义等信息。它描述了表结构的动态变化，使系统可以理解和处理不同的字段和数据类型，而不必直接修改数据库结构。

宽表 (Wide Table)：宽表是在数据库中提前定义了足够多的列，以适应可能出现的各种字段需求，它用来存储不同元信息的实际数据记录（实体）

借鉴salesforce的metadata-driven model
https://architect.salesforce.com/fundamentals/platform-multitenant-architecture

metadata表

tenant_id	entity_type	field_id	filed_name	data_type	column_name

1	Post	1	title	str	column_1
1	Post	2	content	text	column_2
2	Commnent	1	rating	int	column_2
2	Commnent	2	text	text	column_4
2	Product	2	price	int	column_2
2	Product	1	name	str	column_1

data表

tenant_id	entity_id	entity_type	column_1	column_2	column_3	column_4	column_...
1	1	Post	"Title 1"	"Content xx"	NULL	NULL	NULL
1	2	Product	"P Name 1"	1500	NULL	NULL	NULL
2	3	Comment	NULL	5	NULL	"Comment"	NULL

优点：

1. **灵活性高**：通过元数据驱动的设计，可以动态添加和修改实体类型和字段，而无需更改底层数据库结构。

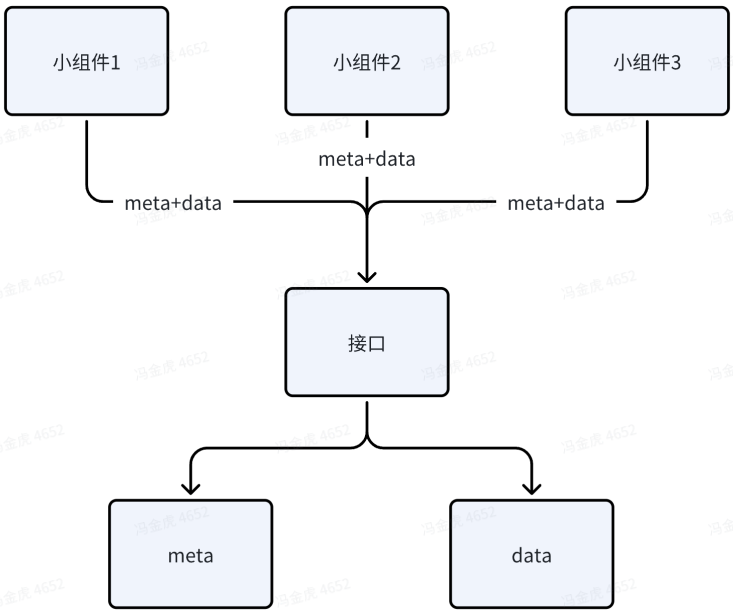
2. **性能较好**：共享表结构避免了 EAV 模型中频繁的 JOIN 操作，提高了查询效率。

缺点：

1. **复杂性增加**：维护元数据表和数据表的映射关系较为复杂，增加了系统管理难度。

2. **数据冗余和存储浪费**：宽表可能包含大量未使用的列，导致存储空间浪费。

3. **扩展受限**：虽然灵活，但某些情况下不如 EAV 模型在处理大量可变字段时那么高效。



文档型数据库

这个说白了就是支持大Json的存储。Mysql/MongoDB/ES都可以一定程度上支持。

类似以下存储结构

ID	Name	Type	Data

1	Product1	wid-1	{
			"Name":"电脑",
			"CPU":"Intel",
			"GPU":"4090"
			}

2	Post1	wid-2	{
			"ID": 1,
			"Content": "hello",
			"IP": "187.163.9.1"
			}

3	Post2	wid-2	{ "ID": 2, "Content": "world",

```
    "IP": "xxx.xxx.xxx.xxx",
    "Sex": "Male"
}
```

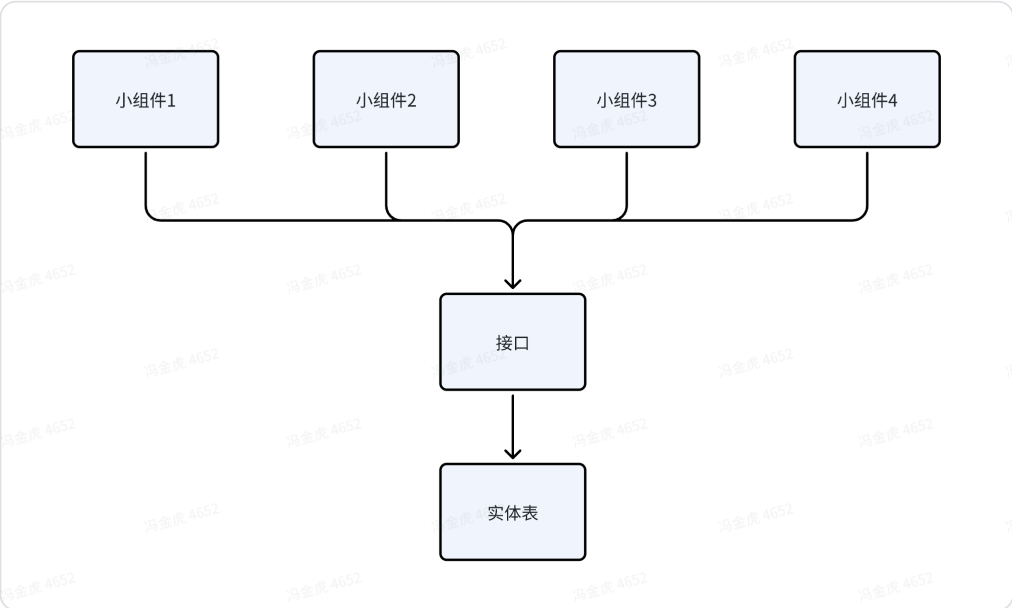
新建小组件：

这里也是新建时没有Schema限制，所以后端不用做任何设计；

更新小组件：

如果还没有记录数据，则无需做任何操作；如果已经有记录，则需要根据具体的变更把相关的数据遍历一遍，按要求修改对应的JSON结构；

某个小组件提交新数据，则直接插入；更新新数据，也是直接替换相应记录。



安全和隔离层面看，每个客户或者每个网站可以单独建一个表。

优点：

使用简单，操作简单

缺点：

性能是灾难，基本无法支持自适应的联表查询

不同研发方式的小组件流程、支持度、特点

这里的研发专指跟小组件存储的网络请求和数据存、取操作相关。

操作：

a. 小组件新建
b. 小组件更新

c. 数据存

d. 数据查取

角色：

a. 内部研发

b. 建站客户

c. 建站站点用户

d. 外部研发

组件维度：

a. 组件类型

b. 组件实例

阶段：

a. 新建页面

b. 更新页面

c. 托管使用页面（使用时）

官方预先生成的组件

特点：人工参与，相对可控

	传统开发法	使用单文件	横向设计法	EAV	Meta driven	文档数据库
前端如何新建小组件	内部研发：按一个小组件类型建一个数据结构、请求接口、参数构造与响应解析；小组件代码按类型存储； 其他角色无法参与	内部研发：按一个小组件类型建一个数据结构、请求接口、参数构造与响应解析的代码；小组件代码按类型存储供检索引用； 其中与后端交互的接口是同一个，除了小组件独有的	内部研发：按一个小组件类型建一个数据结构、请求接口、参数构造与响应解析的代码；小组件代码按类型存储；	内部研发：按一个小组件类型建一个数据结构、请求接口、参数构造与响应解析的代码；小组件代码按类型存储；	内部研发：按一个小组件类型建一个数据结构、请求接口、参数构造与响应解析的代码；小组件代码按类型存储；	内部研发：按一个小组件类型建一个数据结构、请求接口、参数构造与响应解析的代码；小组件代码按类型存储；

		data结构，其他公共参数与响应都一致。				
后端如何新建小组件	内部研发：一个小组件类型对应一个表或者多个表，**手动设计建表**； 一个小组件类型对应**一组增删改查接口**和对应参数与响应； 其他角色无法参与	内部研发：后端提供通用接口，所有小组件类型共用一个接口组；接口参数、返回值都是通用的；也就是写好通用接口和流程后后端就可以不用参与新建了；**没有数据存储建模这一阶段，在流程运行写入时才会有数据建模控制。** 其他角色无法参与；	内部研发：后端提供通用接口，所有小组件类型共用一个接口组；接口参数、返回值都是通用的；也就是写好通用接口和流程后后端就可以不用参与新建了。 其他角色无法参与；	内部研发：后端提供通用接口，设计好3张通用表建模，所有小组件类型共用一个接口组；接口参数、返回值都是通用的；也就是写好通用接口和流程后后端就可以不用参与新建了。 **没有业务数据存储建模这一阶段，在流程运行查询时才会有隐式的建模控制。** 其他角色无法参与；	内部研发：后端提供通用接口，设计好2张通用表建模，所有小组件类型共用一个接口组；接口参数、返回值都是通用的；也就是写好通用接口和流程后后端就可以不用参与新建了。 **没有业务数据存储建模这一阶段，在流程运行写入时才会有隐式的建模控制。** 其他角色无法参与；	内部研发：后端提供通用接口，所有小组件类型共用一个接口组；接口参数、返回值都是通用的；也就是写好通用接口和流程后后端就可以不用参与新建了；**没有数据存储建模这一阶段，在流程运行写入时才会有数据建模控制。** 其他角色无法参与；
前端如何更新小组件类型	内部研发更新小组件库对应类型代码，其他人无法参与	内部研发直接更新组件原型代码	内部研发更新小组件库对应类型代码，其他人无法参与	内部研发更新小组件库对应类型代码，其他人无法参与	内部研发更新小组件库对应类型代码，其他人无法参与	内部研发更新小组件库对应类型代码，其他人无法参与
后端如何更新小组件类型	内部研发更新小组件库对应后端接口和存储部分，其他人无法参与	不用更新	不用更新	不用更新	不用更新	不用更新
wegic建/改页	从建站站点客户的建站chats里提取意图，去小	匹配到小组件原型，实例化小组件实例，	匹配到小组件原型，实例化小组件实例，	匹配到小组件原型，实例化小组件实例，	匹配到小组件原型，实例化小组件实例，	匹配到小组件原型，实例化小组件实例，

面时如何新建小组件实例	组件库向量匹配（如RAG）到合适的小组件类型，嵌入网页区块，新建一个小组件实例；**本质上是使用或实例化，而不是新建。**	与网页代码一起保存	与网页代码一起保存； **同时实例化时前端需要向后端发起数据表结构创建申请，后端接口组中的DDL功能执行数据表创建。**	与网页代码一起保存	与网页代码一起保存 **确定结构后需要申请后端DDL建表操作，后端接口组中的DDL功能执行元信息添加记录（实际上是把DDL转换成元信息表的DML）**	与网页代码一起保存 **确定结构后需要申请后端DDL建表操作，后端新建一个collection** （文档型数据库比较灵活，这里实际上也可以一个组件类型映射一个collection，转换到运行时就是实例化时可以先匹配信息表，看看是否已存在collection，没有则新建，否则使用旧的）
wegic建页面-发布后无历史数据时如何更新小组件实例结构	建站客户只能替换小组件类型，新建一个实例，删除原先的实例，无法更新小组件结构。	小组件实例可以开放修改数据结构的前端接口给建站客户，只需要前端更新数据相关结构并保存即可。	小组件实例可以开放修改数据结构的前端接口给建站客户，客户可以用来修改结构； **前端需要把结构的diff转化为DDL需求向后端发起数据表结构修改申请，后端接口组中的DDL功能执行数据表创建。**	小组件实例可以开放修改数据结构的前端接口给建站客户，客户可以用来修改结构后保存实例代码即可；	小组件实例可以开放修改数据结构的前端接口给建站客户，客户可以用来修改结构后保存实例代码即可； **确定结构后需要申请DDL建表操作，后端接口组中的DDL功能执行元信息修改记录（实际上是把DDL转换成元信息表的DML）**	小组件实例可以开放修改数据结构的前端接口给建站客户，只需要前端更新数据相关结构并保存即可。
wegic页	建站站点客户只能替换小组件类	小组件实例可以开放修改数	小组件实例前端执行结构修	小组件实例前端执行结构修	小组件实例前端执行结构修	小组件实例可以开放修改数

面发布后（产生历史数据）如何更新小组件实例结构	型，新建一个实例，删除原先的实例，无法更新小组件结构。	据结构的前端接口给建站客户，但是需要同时提供处理历史数据的办法，依然在前端处理，后端就是一个文件proxy。 历史数据按照前端的历史数据处理办法处理	改，然后申请后端DDL修改，后端执行后（不一定能成功，部分有数据的DDL修改可能无法成功）将结果给前端；后续前端会按照新数据结构渲染。 历史数据按照新的数据结构格式处理	改即可； 至于后端的数据结构要不要修正，这里我倾向于不修正。如果是修正结构的话要执行批量的DML操作。这里的DML操作既改了结构，也修正了历史数据。 不过即使后端不修，查询时组装返回前端也会按照前端新的结构使用数据的。	改，然后申请后端DDL修改； 后端修改DDL仅修改元信息表记录即可，实际上是DML操作。 历史数据不用修正，因为每次查询都会先从meta表中新建数据Schema.	据结构的前端接口给建站客户，但是需要同时提供处理历史数据的办法，依然在前端处理，后端就是一个proxy。 历史数据按照前端的历史数据处理办法处理。
数据存储特点	1. 数据按小组件类型存储（表组与类型映射） 2. 多租户共享数据库，共享数据表，仅通过租户ID加以区分 3. 只有DML，无DDL	1. 数据按小组件实例存储（文件与实例映射） 2. 多租户天然隔离数据 3. 无数据库操作 4. 数据是孤岛 5. 查询性能低下	1. 数据按小组件实例存储（表组与实例映射） 2. 多租户天然隔离数据 3. 有数据库DDL操作 4. 可以实现较复杂的数据联查或SQL（需提前定义场景协议） 5. 查询性能高	1. 数据全部打平存储，无任何映射 2. 多租户共享数据库，共享数据表，仅通过租户ID加以区分 3. 无任何数据库DDL操作，只有DML操作 4. 存查操作引擎需仔细设计，存储高度灵活	1. 数据物理层面全部打平存储，无任何映射 2. 实际元信息（隐式Schema）是与组件实例映射的 3. 多租户共享数据库，共享数据表，仅通过租户ID加以区分 4. 无任何数据库DDL操作，只有DML操作	1. 数据按照小组件实例存储（也可以按照类型存储） 2. 多租户共享数据库 3. Schema-less, 所有Schema-less的优缺点它都有 4. 性能很差（无索引支持），不支持大数据量 5. 无高级查询支持

				5. 查询性能中低，写入性能中低 6. 无法支持高级SQL查询	5. 存查操作引擎需仔细设计，存储高度灵活 6. 查询性能中高等，写入性能中高等 7. 能执行一定的高级SQL，但是特别高级的也支持不了 8. 唯一索引等东西需要从逻辑层面或者加表支持	
流程控制特点	一个类型一个流程开发，前后端都有高度控制的流程逻辑，支持很复杂的业务逻辑流程控制	后端通用流程，只支持通用的数据增删改查；只能依赖前端做特性数据逻辑处理； **也可以提前做好简单流程库，供组合选择（后面通用，不再赘述）**	后端通用流程，一是DDL流程，二是通用的数据增删改查；三是通过约定协议可做一定的联查	后端通用流程，主要就是EAV映射转换流程，即列转行这个引擎流程	后端通用流程，主要就是把表结构设计到元信息里，找到宽表（数据表）的列映射，实时重建Shema和ORM.	后端通用流程，支持通用的数据增删改查，也能支持简单聚合查询
写入数据（使用时）	提交到小组件类型固定的后端接口，写到类型对应的数据表	后端控制好读写锁，采用append/delete/update模式单条或批量更新，不是全量快照（客户端全量快照的话存在数据一致	直接执行关系型数据库的ORM写入即可	前端提交正常的数据结构，后端需要通过列转行引擎，插入3张固定表。	前端提交正常的数据结构，后端需要通过组件实例ID从元信息记录找到宽表映射列，实时重建Schema, 然后	前端提交正常的数据结构，后端直接写入对应的collection

		性问题），可以类比即时设计的协同编辑，只是不用实时广播。			按照ORM写入数据	
查询数据（使用时）	提交到小组件类型固定的后端接口，从类型对应的数据表读取	后端控制好读写锁，采用全量或者分页返回数据快照	直接执行关系型数据库的ORM查询即可	需要通过EAV映射引擎查询出行，转列，组装成正确的数据结构后返回前端	通过组件实例ID从元信息里找出Schema映射列，然后按照ORM从宽表里取数据	后端直接查询对应的collection
多租户数据隔离方式	共享数据库，共享数据表，仅通过租户ID加以区分	数据库级别隔离	数据表级别隔离	共享数据库，共享数据表，仅通过租户ID加以区分	共享数据库，共享数据表，仅通过租户ID加以区分	数据表级别隔离
数据一致性	数据库正确使用保证了数据的一致性	需要前后端使用非快照+读写锁控制一致性	关系型数据库正确使用保证了数据的一致性	关系型数据库一定程度保证了一致性，但是历史数据的一致性很难保证	数据一致性要后端保障，需要设计特殊的规则来支撑	数据一致性要后端保障，通过文档型数据库操作可以有基本保障
安全性	多租户仅通过租户ID加以区分数据，有一定安全隐患； 现实必须要解决的一个前提问题：多租户的用户查询一个租户页面的小组件数据的时候，必须要有鉴权措施，否则可以随意拉取到别的租户的页面数据（只需暴力尝试不同页面ID或者小组件ID作为参数）	1. 多租户的数据间是安全隔离的 2. 需要对组件实例鉴权，以防越页面取数据	1. 多租户的数据间是安全隔离的 2. 需要对组件实例鉴权，以防越页面取数据 3. 支持一些行和列级别的数据筛选，不过需要提前定好查询协议 4. 数据库DDL操作，需要做好严格的验证监控	1. 多租户仅通过租户ID加以区分数据，有一定安全隐患； 2. 需要对组件实例鉴权，以防越页面取数据 3. 无数据库DDL操作，仅有DML操作	1. 多租户仅通过租户ID加以区分数据，有一定安全隐患； 2. 需要对组件实例鉴权，以防越页面取数据 3. 无数据库DDL操作，仅有DML操作	1. 多租户的数据间是表级别安全隔离的 2. 需要对组件实例鉴权，以防越页面取数据 3. 有建表操作，无更新表操作 4. 支持一些行和列级别的数据筛选，不过需要提前定好查询协议

		3. 很难做查询和行列级别数据筛选，因为后端只有通用取数逻辑，所以会把整个小组件实例的数据全部下发到前端，前端选择性展示，这方面可能存在数据隐患		4. 支持一些行和列级别的数据筛选，不过需要提前定好查询协议	4. 支持一些行和列级别的数据筛选，不过需要提前定好查询协议	
结论	1. 只有内部研发可以生成和更新，其他人只能使用 2. 复杂的且有必要价值的小组件可以用这种方式做到组件库，供生成使用 3. 只从小组件使用上来看应该是体验最好的	1. 后端通用，意味着投入很小就可以支持绝大多数的简单存取需求 2. 对复杂存储和查询功能基本不支持，遇到这种场景需要pass掉此方案	1. 后端相对通用 2. 一些协议需要提前约定 3. 支持一定程度的复杂查询 4. 有DDL操作，既有新建也有更改 5. 表数和组件实例数正相关，会有无限膨胀的风险	1. 后端相对通用 2. EAV引擎复杂度高，需要仔细设计和开发 3. 无DDL操作 4. 表数固定，仅数据膨胀 5. 运行效率较低	1. 后端相对通用 2. 运行时元信息重建和查询复杂度高，需要仔细设计和研发 3. 无DDL操作 4. 表数固定，仅数据膨胀 5. 运行效率尚可	1. 后端相对通用 2. 有DDL建表操作 3. 表数和组件实例数正相关，会有无限膨胀的风险 4. 运行效率低下

LLM辅助生成

特点：全/半自动化，需要仔细审核上述流程中LLM需要做到什么步骤，额外增加哪些工作和代价

此时角色打平（内部研发、 建站客户），只有创建者这一角色。

LLM辅助生成可以认为有两种形式，一是全部是新的，二是基于组件库来修改，下面的几个论述流程实际上已经覆盖了这两种情形，只是交互层面提供入口即可。

	传统开发法	使用单文件	横向设计法	EAV	Meta driven	文档数据库
前端如何新建小组件类型	创建者提出Prompt, LLM返回小组件前端代码，和对应的后端接口控制模块	创建者提出Prompt, LLM返回小组件前端代码，和对应的后端接口控制模块； Prompt需要包含以下方面： 1. 与后端交互的接口是同一个，除了小组件独有的data结构，其他公共参数与响应都一致，需要嵌入这些参数 2. 全量的data数据处理逻辑。 然后LLM生成的小组件前端代码保存到库里即可	创建者提出Prompt, LLM返回小组件前端代码，和对应的后端接口控制模块； Prompt需要包含以下内容： 1. 生成数据表DDL设计的指令 2. Prompt返回值结构里包含数据表DDL 3. 这个值需要和小组件类型代码保存在一块，以供实例化时使用 当然也可以不用生成DDL, 而是和人工生成一样，被动做结构转换，运行时建表	创建者提出Prompt, LLM返回小组件前端代码，和对应的后端接口控制模块；其他的不需要了	创建者提出Prompt, LLM返回小组件前端代码，和对应的后端接口控制模块；其他的不需要了	创建者提出Prompt, LLM返回小组件前端代码，和对应的后端接口控制模块；其他的不需要了
后端如何新建小组件类型	无法实现，现在让LLM控制后端代码和数据库，基本无法实现	后端通用接口，无LLM参与	后端通用接口，无LLM参与；	后端通用接口，无LLM参与	后端通用接口，无LLM参与；	后端通用接口，无LLM参与
前端如何更新小组	——	Prompt修改组件前端代码，然后保存	Prompt修改组件前端代码，然后保存，包含DDL部分内容	Prompt修改组件前端代码，然后保存，已经包含DDL部分内容	Prompt修改组件前端代码，然后保存	Prompt修改组件前端代码，然后保存

件类型						
后端如何更新小组件类型	——	不用更新	不用更新	不用更新	不用更新	不用更新
wegic建/改页面时如何新建小组件实例	——	1. 匹配到小组件原型，实例化小组件实例，与网页代码一起保存 2. 生成页面时，直接新生成一个小组件实例 这种时候Prompt包含的东西与上面生产组件类型时要求一致。	1. 匹配到小组件原型，实例化小组件实例，与网页代码一起保存； **同时实例化时前端需要向后端发起（LLM给的）DDL数据表结构创建申请，后端接口组中的DDL功能执行数据表创建。** 2. 生成页面时，直接新生成一个小组件实例，同时执行DDL数据表创建申请，**后端接口组中的DDL功能执行数据表创建**	1. 匹配到小组件原型，实例化小组件实例，与网页代码一起保存 2. 生成页面时，直接新生成一个小组件实例	1. 匹配到小组件原型，实例化小组件实例，与网页代码一起保存 **确定结构后需要申请后端DDL建表操作，后端接口组中的DDL功能执行元信息添加记录（实际上是把DDL转换成元信息表的DML）** 2. 生成页面时，直接新生成一个小组件实例，此时也需要**确定结构后需要申请后端DDL建表操作，后端接口组中的DDL功能执行元信息添加记录（实际上是把DDL转换成元信息表的DML）**	1. 匹配到小组件原型，实例化小组件实例，与网页代码一起保存 **确定结构后需要申请后端DDL建表操作，后端新建一个collection** （文档型数据库比较灵活，这里实际上也可以一个组件类型映射一个collection，转换到运行时就是实例化时可以先匹配信息表，看看是否已存在collection，没有则新建，否则使用旧的） 和无LLM类似。

wegic建页面-发布后无历史数据时如何更新小组件实例结构	——	小组件实例可以开放修改数据结构的前端接口给建站客户，通过Prompt让LLM按规则修改	小组件实例可以开放修改数据结构的前端接口给建站客户，客户可以通过LLM修改结构； **前端需要把结构的diff转化为DDL需求向后端发起数据表结构修改申请，后端接口组中的DDL功能执行数据表修改。**	小组件实例可以开放修改数据结构的前端接口给建站客户，客户可以通过LLM用来修改结构后保存实例代码即可；	小组件实例可以开放修改数据结构的前端接口给建站客户，客户可以通过LLM来修改结构后保存实例代码即可； **确定结构后前端需要申请DDL建表操作，后端接口组中的DDL功能执行元信息修改记录（实际上是把DDL转换成元信息表的DML）** 和无LLM参与类似	小组件实例可以开放修改数据结构的前端接口给建站客户，只需要前端更新数据相关结构并保存即可。 和无LLM类似。
wegic页面发布后（产生历史数据）如何更新小组件实例结构	——	Prompt里需包含数据转换或者历史数据处理逻辑	小组件实例前端执行结构修改，然后申请后端DDL修改，后端执行后（不一定能成功，部分有数据的DDL修改可能无法成功）将结果给前端；后续前端会按照新数据结构渲染。 历史数据按照新的数据结构格式处理。	小组件实例前端执行结构修改即可； 至于后端的数据结构要不要修正，这里我倾向于不修正。如果是修正结构的话要执行批量的DML操作。这里的DML操作既改了结构，也修正了历史数据。 不过即使后端不修，查询时组装返回前端也会按照前端新的结构使用数据的。	小组件实例前端执行结构修改，然后申请后端DDL修改； 后端修改DDL仅修改元信息表记录即可，实际上是DML操作。 历史数据不用修正，因为每次查询都会先从meta表中新建数据Schema. 和无LLM参与类似	小组件实例可以开放修改数据结构的前端接口给建站客户，但是需要同时提供处理历史数据的办法，依然在前端处理，后端就是一个proxy。 历史数据按照前端的历史数据处理办法处理。 和无LLM类似。

数据存储特点	——	1. 数据按小组件实例存储（文件与实例映射） 2. 多租户天然隔离数据 3. 无数据库操作 4. 数据是孤岛 5. 查询性能低下	1. 数据按小组件实例存储（表组与实例映射） 2. 多租户天然隔离数据 3. 有数据库DDL操作 4. 可以实现较复杂的数据联查或SQL（需提前定义场景协议） 5. 查询性能高	1. 数据全部打平存储，无任何映射 2. 多租户共享数据库，共享数据表，仅通过租户ID加以区分 3. 无任何数据库DDL操作，只有DML操作 4. 存查操作引擎需仔细设计，存储高度灵活 5. 查询性能中低，写入性能中低 6. 无法支持高级SQL查询	1. 数据物理层面全部打平存储，无任何映射 2. 实际元信息（隐式Schema）是与组件实例映射的 3. 多租户共享数据库，共享数据表，仅通过租户ID加以区分 4. 无任何数据库DDL操作，只有DML操作 5. 存查操作引擎需仔细设计，存储高度灵活 6. 查询性能中高等，写入性能中高等 7. 能执行一定的高级SQL，但是特别高级的也支持不了 8. 唯一索引等东西需要从逻辑层面或者加表支持	1. 数据按照小组件实例存储（也可以按照类型存储） 2. 多租户共享数据库 3. Schema-less, 所有Schema-less的优缺点它都有 4. 性能很差（无索引支持），不支持大数据量 5. 无高级查询支持
流程控制特点	——	后端通用流程，只支持通用的数据增删改查；只能依赖前端做特性	后端通用流程，一是DDL流程，二是通用的数据增删改查；三是	后端通用流程，主要就是EAV映射转换流程，即列转行这个引擎流程	后端通用流程，主要就是把表结构设计到元信息里，找到宽表（数	后端通用流程，支持通用的数据增删改查，也能支持简单聚合查询

		数据逻辑处理； **也可以提前做好简单流程库，供组合选择（后面通用，不再赘述）**	通过约定协议可做一定的联查		据表）的列映射，实时重建Shema和ORM.	
写入数据（使用时）	——	后端控制好读写锁，采用append/delete/update模式单条或批量更新，不是全量快照（客户端全量快照的话存在数据一致性问题），可以类比即时设计的协同编辑，只是不用实时广播。	直接执行关系型数据库的ORM写入即可	前端提交正常的数据结构，后端需要通过列转行引擎，插入3张固定表。	前端提交正常的数据结构，后端需要通过组件实例ID从元信息记录找到宽表映射列，实时重建Schema, 然后按照ORM写入数据	前端提交正常的数据结构，后端直接写入对应的collection
查询数据（使用时）	——	后端控制好读写锁，采用全量或者分页返回数据快照	直接执行关系型数据库的ORM查询即可	需要通过EAV映射引擎查询出行，转列，组装成正确的数据结构后返回前端	通过组件实例ID从元信息里找出Schema映射列，然后按照ORM从宽表里取数据	后端直接查询对应的collection
多租户数据隔离方式	——	数据库级别隔离	数据表级别隔离	共享数据库，共享数据表，仅通过租户ID加以区分	共享数据库，共享数据表，仅通过租户ID加以区分	数据表级别隔离
数据一致性	——	需要前后端使用非快照+读写锁控制一致性	关系型数据库正确使用保证了数据的一致性	关系型数据库一定程度保证了一致性，但是历史数据的一致性很难保证	数据一致性要后端保障，需要设计特殊的规则来支撑	数据一致性要后端保障，通过文档型数据库操作可以有基本保障
	——					

安全性		1. 多租户的数据间是安全隔离的 2. 需要对组件实例鉴权，以防越页面取数据 3. 很难做查询和行列级别数据筛选，因为后端只有通用取数逻辑，所以会把整个小组件实例的数据全部下发到前端，前端选择性展示，这方面可能存在数据隐患	1. 多租户的数据间是安全隔离的 2. 需要对组件实例鉴权，以防越页面取数据 3. 支持一些行和列级别的数据筛选，不过需要提前定好查询协议 4. 数据库DDL操作，需要做好严格的验证监控	1. 多租户仅通过租户ID加以区分数据，有一定安全隐患； 2. 需要对组件实例鉴权，以防越页面取数据 3. 无数据库DDL操作，仅有DML操作 4. 支持一些行和列级别的数据筛选，不过需要提前定好查询协议	1. 多租户仅通过租户ID加以区分数据，有一定安全隐患； 2. 需要对组件实例鉴权，以防越页面取数据 3. 无数据库DDL操作，仅有DML操作 4. 支持一些行和列级别的数据筛选，不过需要提前定好查询协议	1. 多租户的数据间是表级别安全隔离的 2. 需要对组件实例鉴权，以防越页面取数据 3. 有建表操作，无更新表操作 4. 支持一些行和列级别的数据筛选，不过需要提前定好查询协议
结论	无法实现	1. 后端通用，意味着投入很小就可以支持绝大多数的简单存取需求 2. 对复杂存储和查询功能基本不支持，遇到这种场景需要pass掉此方案	1. 后端相对通用 2. 一些协议需要提前约定 3. 支持一定程度的复杂查询 4. 有DDL操作，既有新建也有更改 5. 表数和组件实例数正相关，会有无限膨胀的风险	1. 后端相对通用 2. EAV引擎复杂度高，需要仔细设计和开发 3. 无DDL操作 4. 表数固定，仅数据膨胀 5. 运行效率较低	1. 后端相对通用 2. 运行时元信息重建和查询复杂度高，需要仔细设计和研发 3. 无DDL操作 4. 表数固定，仅数据膨胀 5. 运行效率尚可	1. 后端相对通用 2. 有DDL建表操作 3. 表数和组件实例数正相关，会有无限膨胀的风险 4. 运行效率低下

开放的开发接口

选型中要更重视安全性，然后开放给出相应接口并给出严格限制即可支持功能。

选型总结

	传统开发法	使用单文件	横向设计法	EAV	Meta driven	文档数据库
选型决策成本粗估	1. 后端开发成本：5（非常高，所以得分低，预估中等复杂度1天一个） 2. 前端开发成本：2（相对来说较高，但不一定比LLM生成慢） 3. LLM调试成本：暂时无法支持 4. 运行效率曲线：5（满分，基本遇到效率问题都会被解决） 5. 效果： - [ ] 每个小组件单独开发(前后端) - [ ] 对需求支持度最高 - [ ] 可以用于构建复杂组件库	1. 后端开发成本：3（预估3天，主要是与文件系统对接，分布式锁逻辑防并发问题） 2. 前端开发成本：2（需要自己处理数据逻辑） 3. LLM调试成本：4（全在前端，需要理解接口规则，处理数据分页展示等逻辑，并前端LLM Prompt 自动化操作） 4. 运行效率曲线：1（速度比较差，不支持复杂查询） 5. 效果： - [ ] 后端只有一次性的工作，后续前端调用	1. 后端开发成本：3（预估3天, 处理自动化DDL） 2. 前端开发成本：2（需要自己处理数据逻辑） 3. LLM调试成本：3（需要理解接口规则，还可以做分页、筛选等操作，并前端LLM Prompt 自动化操作） 4. 运行效率曲线：4（速度尚可，支持一定程度的复杂查询） 5. 效果： - [ ] 后端只有一次性的工作，后续前端调用	1. 后端开发成本：4（预估5天，要实现EAV查询引擎与严格自测） 2. 前端开发成本：2（需要自己处理数据逻辑） 3. LLM调试成本：3（全在前端，需要理解接口规则，并前端LLM Prompt 自动化操作） 4. 运行效率曲线：2（速度比较差，较多的联表和多行操作，不支持复杂查询） 5. 效果：	1. 后端开发成本：4（预估4天，实现字段的meta表和data表映射转换引擎） 2. 前端开发成本：2（需要自己处理数据逻辑） 3. LLM调试成本：3（全在前端，需要理解接口规则，并前端LLM Prompt 自动化操作） 4. 运行效率曲线：4（速度较好，支持一般复杂查询） 5. 效果：	1. 后端开发成本：3（预估3天） 2. 前端开发成本：2（需要自己处理数据逻辑） 3. LLM调试成本：3（全在前端，需要理解接口规则，并前端LLM Prompt 自动化操作） 4. 运行效率曲线：1（速度比较差，不支持复杂查询） 5. 效果： - [ ] 后端只有一次性的工作，后续前端调用

		▢ 复杂操作不支持，只支持基本的增删改查 ▢ 运行效率偏低，不能承担大数据量（拍脑袋：5000行）	▢ 支持基本的增删改查和一定程度的复杂查询 ▢ 运行效率中高，表可以加索引，所以效率比较好 ▢ 天然支持并发处理，单个小组件存到几十万数据量都可以存 ▢ 运维不友好，要做DDL操作，需要要到账号权限	▢ 后端只有一次性的工作，后续前端调用 ▢ 支持基本的增删改查，也支持列过滤和分页 ▢ 运行效率偏低，能承担一定量的数据查询，但是全部总量有限制，比如50万行记录后，需要想办法分库分表等优化操作 ▢ 运维友好，无任何DDL操作	▢ 后端只有一次性的工作，后续前端调用 ▢ 支持基本的增删改查和一定程度的复杂查询，如果需要支持唯一性等特殊操作，得独立开发辅助模块；不支持多字段排序功能；	▢ 复杂操作不支持，只支持基本的增删改查 ▢ 运行效率偏低，不能承担大数据量（拍脑袋：全部数据50万行）

					☐ 运行效率较高，加上表的路由索引（可手动提前建），总量不太受限制；单表总量不能超过1000万行 ☐ 运维友好，无任何DDL操作	☐ 一种是DDL，需要建表，但不用改表，查询效率一般（这种就不如横向法效率好，缺点却一样多）；一种是无DDL，查询效率极低 ☐ 不支持改动已产生数据的字段名

总结：

支持LLM自动生成和开放性接口的方式，需要综合对LLM的支持，考虑到后端应该稳定而不变更（LLM做不到），同时支持一定的运行效率，运维友好性等，可以选用Meta driven的方式， 除了不能实现较复杂的查询，从开发的难度、LLM通用性和查询效率以及支持的数据量上看，都可以支撑我们的初始阶段需要。后续如果满足不了需求，也可以再迁移；