项目性能与稳定性优化白皮书（新手解读版）

> *文档目标：让未来的我一眼看懂，这个 Agent/RAG 系统为什么要做这些优化、每一步防住了什么问题、系统出故障时会怎么兜底。*
> *适合读者：刚接触并发、缓存、连接池、熔断、RAG 服务的新手。*
> *一句话总结：这轮优化不是为了“把 QPS 数字吹高”，而是为了让系统在高并发和外部依赖故障下，不乱卡、不乱崩、能解释、能降级。*

最终 QPS 与 Token/请求汇总

场景	是否调用 LLM	测试口径	最终 QPS	Q/10s	p95	Token/请求	结论
ASK 缓存命中	否	/ask，缓存命中，50/10	187.83	1878	104ms	0 增量 token	高频重复问答走缓存，吞吐最高
ASK 缓存命中高并发	否	/ask，缓存命中，200/50	153.82	1538	804ms	0 增量 token	并发升高后长尾变明显
ASK miss 稳定慢路径	是	关闭缓存，并发 12	1.49	14.9	14.3s	约 2370	可用但长尾已高
ASK miss 极限吞吐	是	关闭缓存，并发 20	2.13	21.3	13.5s	约 2121	本轮最高 ASK miss QPS
ASK miss 过饱和	是	关闭缓存，并发 24	1.91	19.1	15.7s	约 1958	QPS 回落，已过饱和点
Agent 规则工单	否	/agent 规则查单	受限流影响	-	34ms token 样本	0	规则路由不消耗 LLM token
工单详情直查	否	/tickets/{id}，500/50	154.59	1546	830ms	0	代表普通工单读路径吞吐
Agent Planner 稳定点	是	LLM Planner，并发 32	10.39	103.9	4.7s	未暴露	更推荐作为稳定工作点
Agent Planner 极限吞吐	是	LLM Planner，并发 48	11.07	110.7	6.27s	未暴露	本轮最高 Agent Planner QPS
Agent Planner 过饱和	是	LLM Planner，并发 64	7.34	73.4	13.6s	未暴露	QPS 回落，出现 1 个 500

简化版核心表格：

核心链路	测试口径	并发	QPS	P95 延迟	平均首 Token	Tokens / 请求	结论
RAG 问答，缓存不命中	/ask/stream，唯一问题，真实检索 + LLM 生成	20	**2.13**	**13.46s**	**用户体感 5.61s / LLM 1.16s**	**约 2,121**	主要瓶颈在检索长尾 + LLM 生成

Agent 工单，LLM Planner	/agent，建单类请求，强制走 LLM Planner	32	**10.39**	**4.72s**	不适用	暂未暴露精确值	32 并发是稳定展示点，64 并发开始退化
RAG 问答，缓存命中	/ask，重复问题命中缓存	50	**153.82**	**804ms**	接近 0 / 无需 LLM	**0**	缓存命中绕过检索和 LLM
工单规则路径	/agent，查单/催办等规则可处理请求	50	可补测	通常低于 LLM Planner	不适用	**0**	确定性操作不走 LLM，适合高频请求

0. 先讲人话：这个系统为什么会慢、会崩

这个项目是一个 FastAPI + RAG + Agent 服务。一次用户提问大概会经过这些环节：

```text
浏览器/客户端
  -> FastAPI 鉴权
  -> Redis 缓存
  -> Embedding 模型
  -> Qdrant 向量检索
  -> LLM 生成答案
  -> PostgreSQL 落库/审计
  -> 返回给用户
```

这里面任何一层慢，都可能让用户觉得“系统卡了”。更麻烦的是，高并发下瓶颈会互相传染：

- Redis 慢：缓存和鉴权都可能慢。
- PostgreSQL 连接不够：请求排队，线程被占住。
- Qdrant 慢：RAG 查不到证据，首字时间变长。
- LLM 慢：流式响应迟迟不出第一个 token。
- Python CPU 忙：即使数据库没问题，请求也会排队。

所以优化的核心不是“哪里慢就盲目加机器”，而是：

1. 先打点，看慢在哪里。
2. 再按依赖层级压测，别一上来压全链路。
3. 找到瓶颈后，做最小有效优化。
4. 对外部依赖加兜底，避免一个服务挂了拖死所有请求。

模块	优化前测试结果	优化后测试结果	当前结论
API 入口 /health	小测 5/5 成功，p95≈16ms	1000/100 下 1000/1000 成功，p95≈198ms	入口本身不是主要瓶颈
API 查单读路径	lookup 1000/100： 30 成功，970 超时， p95≈15107ms	lookup 1000/100： 423 成功，577 超时， p95≈15061ms； 300/30、500/50 均 100% 成功	有明显改善，但 100 并发读路径仍未彻底解决
建单写路径	create 600/60： 0 成功，600 失败， p95≈15064ms	create 600/60： 593 成功，7 失败， p95≈927ms	从完全雪崩优化到基本可用
单工单催办	escalate 600/60： 17 成功，583 失败， p50≈15029ms	escalate 600/60： 600 成功，0 失败， p95≈1238ms	原 60 并发雪崩点已越过

多工单分散更新	escalate_many 600/60/20：58 成功，542 失败，p95≈15023ms	escalate_many 600/60/20：600 成功，0 失败，p95≈502ms	分散写路径改善最明显
ASK 缓存命中	同一问题 第一次约 28.6s 第二次约 24ms	本轮 cache hit 40/40，retrieve=0ms，answer=0ms	命中缓存后确实绕过检索和 LLM
RAG 检索策略	dense R@5=90.00%，p50≈211ms	hybrid_rrf R@5=96.15%，p50≈216ms	召回明显提升，延迟代价很小
Qdrant 故障兜底	外部依赖挂时可能拖住主链路	单测熔断通过，但 docker stop qdrant 后真实请求仍返回 500	设计有了，真实 chaos 还要修

1. 打点计时：先让系统会说“我慢在哪里”

测试场景	ask 缓存命中，20 并发，40 请求	ask 缓存 miss，3 并发，3 请求	/ask/stream 热态问题
总耗时 / wall	p50≈184ms,p95≈440ms	p50≈1669ms,p95≈5741ms	完整 answer≈2566ms
auth	p50≈4ms,p95≈12m	未单列	未单列
cache_lookup	p50≈0ms,p95≈2ms,max≈269ms	p50≈0ms，p95≈1ms	未单列
retrieve	0ms	p50≈481msp95≈4297ms	retrieve_done ≈194ms
answer	0ms	p50≈1168ms，p95≈1414ms	TTFT≈875ms
persist_bundle	p50≈19ms，p95≈306ms	p50≈19ms，p95≈20ms	未单列
说明	命中缓存后检索和 LLM 被绕过，但持久化仍有长尾	miss 长尾主要在 retrieve，不是缓存查询	首 token 可观测，能区分检索慢还是 LLM 首包慢

防止什么问题

新手最容易犯的错是只看 QPS 和 P95，然后猜瓶颈。

比如 `/ask` 慢了，可能是：

- 鉴权慢。
- Redis 缓存 miss。
- Embedding 慢。
- Qdrant 检索慢。
- LLM 首 token 慢。
- PostgreSQL 落库慢。

如果没有分段耗时，只能瞎猜。

做了什么动作

增加了统一请求计时中间件和阶段打点：

- `src/api/request_timing.py`
- FastAPI middleware 自动统计 `http_total`
- ASK 链路记录：
  - `auth`
  - `cache_lookup`

- `retrieve`
- `answer`
- `persist_bundle`
- `response_build`

流式接口 `/ask/stream` 还记录了：

- `retrieve_done`
- `first_token`
- `ttft_ms`
- `stream_tokens`

出问题时会怎么处理

响应 meta 和日志里可以看到阶段耗时。

例如流式返回里出现：

```text
event: status
data: {"stage":"retrieve_done","latency_ms":218}

event: status
data: {"stage":"first_token","ttft_ms":832}
```

意思是：

- 检索花了 218ms。
- 检索结束后，到第一个 token 出来花了 832ms。

这样就能判断：

- 如果 `retrieve` 高，是 Embedding/Qdrant 问题。
- 如果 `ttft` 高，是 LLM 首 token 慢。
- 如果 `persist_bundle` 高，是 PostgreSQL 写审计拖慢。

新手解释

打点就像给流水线每个工位放秒表。以前只知道“饭上晚了”，现在能知道是“点餐慢、后厨慢、还是上菜慢”。

2. 压测顺序：按金字塔原则，不要一上来压全链路

层级	场景	优化前结果	优化后结果	结论
API 入口	/health	小测 p95≈16ms，5/5 成功	1000/100：1000 成功，0 失败，p95≈198ms	入口不是首要瓶颈
DB 读	lookup 300/30	无旧同口径	300 成功，0 失败，p95≈271ms	30 并发读稳定
DB 读	lookup 500/50	无旧同口径	500 成功，0 失败，p95≈256ms	50 并发读稳定
DB 读	lookup 1000/100	30 成功，970 失败，p95≈15107ms	423 成功，577 失败，p95≈15061ms	有改善，但 100 并发仍不稳定
DB 写	create 600/60	0 成功，600 失败，p95≈15064ms	593 成功，7 失败，p95≈927ms	写路径优化非

				常明显
DB 写	escalate 600/60	17 成功，583 失败，p50≈15029ms	600 成功，0 失败，p95≈1238ms	热点更新雪崩点已越过
DB 写	escalate_many 600/60/20	58 成功，542 失败，p95≈15023ms	600 成功，0 失败，p95≈502ms	分散写路径稳定性明显提升
MCP	MCP lookup 1000/100	1000 成功，0 失败，p95≈4258ms	未复测	MCP 稳但慢，后续单独优化
ASK 缓存	/ask cache hit 20 并发	历史同问第二次约 24ms	40/40 命中，retrieve=0ms，answer=0ms	缓存命中能绕过 RAG 和 LLM
ASK 检索	/ask cache miss 3 并发	未形成统一旧口径	retrieve p95≈4297ms，answer p95≈1414ms	miss 长尾主要在 retrieve
流式 LLM	/ask/stream 热态	原先只看总耗时	retrieve≈194ms，TTFT≈875ms，answer≈2566ms	用户能先看到 token

防止什么问题

如果直接压 `/ask` 或 `/orders` 这种全链路接口，慢了也不知道是谁的问题。

全链路里混了：

- 鉴权
- Redis
- PostgreSQL
- Qdrant
- LLM
- JSON 序列化
- 审计落库

结果就是：看起来所有东西都慢，实际上无法定位。

做了什么动作

采用“依赖从底到顶”的压测顺序：

```text
1. /health
2. 纯 DB 读接口
3. DB 写接口
4. MCP 工具接口
5. ASK 缓存命中路径
```

6. ASK 检索路径
7. ASK 流式 LLM 全链路
```

### 出问题时会怎么处理

按层定位：

- `/health` 慢：API worker 或 event loop 有问题。
- DB 读慢：PostgreSQL 连接池或 SQL 有问题。
- DB 写慢：锁、审计、连接池有问题。
- ASK 缓存命中还慢：鉴权、Redis、Python/GIL 可能有问题。
- 检索慢：Embedding/Qdrant 问题。
- LLM 慢：模型服务、网络、供应商限流问题。

### 新手解释

这就像查家里断电。不要一上来拆电视，先看总闸，再看插座，再看电器。

## 3. 监控三大件：别只看 QPS

压测工具给的 QPS、P95 是“症状”，真正的病因要看三类指标。

| 监控对象 | 关键指标 | 异常说明 |
| --- | --- | --- |
| Redis | connected_clients、latency、blocked_clients | 客户端暴涨说明连接池 / Worker 配置要看；latency 毛刺可能是 Redis 持久化或阻塞命令；blocked_clients>0 要重点排查阻塞命令 |
| PostgreSQL | pg_stat_activity、temp_files、连接池等待 | 活跃连接接近上限说明连接池排队；temp_files 说明 SQL 可能没走索引、发生磁盘排序 |
| Python 进程 | CPU、线程、Worker 数、event loop 心跳 | CPU 90%+ 时，加 DB 连接池没用，瓶颈可能是 Python/GIL/序列化/Embedding |

本节测试数据：高压后系统快照

| 监控对象 | 本轮压测后观测 | 解读 |
| --- | --- | --- |
| API health | {"status":"ok","stage":"l2"} | 高压后 API 仍能响应 |
| PostgreSQL activity | active=1，idle=10 | 没观察到连接池被完全拖死 |
| PostgreSQL temp files | 0 | 没观察到磁盘临时排序 |
| Redis clients | connected_clients=37<br>blocked_clients=0 | Redis 客户端较多，但没有阻塞客户端 |

常用命令：

```bash
docker compose exec redis redis-cli INFO clients
docker compose exec redis redis-cli --latency

docker compose exec postgres psql -U policy -d policy_kb -c "select state, count(*) from pg_stat_activity group by state;"
docker compose exec postgres psql -U policy -d policy_kb -c "select datname, temp_files, temp_bytes from pg_stat_database;"

docker compose exec api sh -c "ps -ef | grep '[u]vicorn'"
top -H
```
```

```

## 4. Worker 与连接池：不是越多越快

### 防止什么问题

一开始很容易以为：

```text
Worker 越多，并发越强。
连接池越大，数据库越快。
```

这是错的。

如果机器 CPU 核数有限，Worker 过多会导致：

- 进程切换开销变大。
- 每个 Worker 都加载模型，内存暴涨。
- 每个 Worker 都有自己的 DB/Redis 连接池，总连接数翻倍。

做了什么动作

固定生产默认：

```bash
API_WORKERS=2
DB_POOL_SIZE=5
DB_MAX_OVERFLOW=5
DB_POOL_TIMEOUT_SECONDS=30
REDIS_MAX_CONNECTIONS=20
```

API 启动命令默认：

```bash
uvicorn src.api.app:app \
  --host 0.0.0.0 \
  --port 8080 \
  --workers 2 \
  --log-level warning
```

配置项	本轮复测值	说明
API_WORKERS	2	当前生产建议值，避免 4 worker 带来过多模型 / 连接池副本
DB_POOL_SIZE	5	每个 worker 的基础连接数
DB_MAX_OVERFLOW	5	每个 worker 的溢出连接数
API 理论最大 DB 连接	2 × (5+5)=20	还要额外考虑 MCP 等其他服务连接
REDIS_MAX_CONNECTIONS	20	限制单进程 Redis 连接数

实验	观测结果	结论
单 worker vs 4 worker，并发 20	效率比 11.2x -> 7.4x；墙钟约 0.24s -> 0.14s	多 worker 有帮助，但不是线性提升
/health 扛 30 并发缓存 ASK	p50≈10ms，p95≈11ms，max≈	event loop 没被整体冻住
```

	16ms	
高压后 PostgreSQL activity	`active=1, idle=10`	当前 2 worker + pool 5+5 没把连接完全打死

要理解的关键点

连接池是“每个 Worker 一份”，不是全局一份。

当前 API 最大 PostgreSQL 连接数约等于：

2 workers * (pool_size 5 + max_overflow 5) = 20

MCP 也有自己的连接池，所以还要额外算。

出问题时会怎么处理

如果 PostgreSQL 活跃连接经常打满：

- CPU 不高：可以小幅加 `DB_POOL_SIZE`。
- CPU 很高：不要加连接池，先优化代码或减少 Worker。
- 连接数太多：减少 `API_WORKERS` 或 `DB_MAX_OVERFLOW`。

新手解释

Worker 像收银台，连接池像刷卡机。收银台太多但刷卡机不够，大家还是排队；收银台太多还会挤成一团。

5. 鉴权优化：干掉每次请求查用户表

防止什么问题

场景	优化前	优化后	结论
lookup 1000/100	30 成功，970 超时，p95 ≈15107ms	423 成功，577 超时，p95 ≈15061ms	成功数提升约 14.1 倍，但 100 并发仍有大量超时
lookup 300/30	无旧同口径	300 成功，0 失败，p95≈271ms	30 并发稳定
lookup 500/50	无旧同口径	500 成功，0 失败，p95≈256ms	50 并发稳定

原先每次请求都可能做：

JWT 解码 -> get_user_by_id -> 查 PostgreSQL users 表

高并发时，这会让“每个请求都先抢 DB 连接”。即使业务只是读缓存，也会被鉴权查库拖慢。

做了什么动作

改成：

```
/login 或 /register 成功
  -> 签发 JWT
  -> 顺便把用户快照写入 Redis Hash

后续请求
  -> 解 JWT
  -> 从 Redis Hash 读用户快照
  -> 挂到 request.state
  -> 不再每次 get_user_by_id 查 DB
```

Redis Hash 结构类似：

auth:user:<user_id>
id
username
role
is_active
email
phone

用 Hash 的好处是可以单独改字段：

HSET auth:user:<user_id> is_active 0

不需要重写整个 JSON 对象。

出问题时会怎么处理

正常情况：

- Redis 有用户快照：通过。
- Redis 快照里 `is_active=0`：拒绝，返回 403。
- Redis miss：拒绝，返回 401。

Redis 故障时：

1. 先尝试当前 Worker 的本地短 TTL 快照。
2. 如果本地也没有，走 JWT-only fallback。
3. 响应头标记

X-Auth-Fallback: true
X-Auth-Fallback-Mode: local_snapshot

或：

X-Auth-Fallback: true
X-Auth-Fallback-Mode: jwt_only

为什么要 JWT-only fallback

Redis 状态	本地快照	处理方式	当前验证
Redis 正常，快照 active	有 Redis 用户快照	放行	登录 / 鉴权链路已改
Redis 正常，`is_active=0`	不依赖本地	返回 403	逻辑约束已实现
Redis 正常，但 miss	不依赖本地	返回 401，要求重新登录	逻辑约束已实现
Redis 异常	当前 worker 有本地短 TTL 快照	放行，并标记 `X-Auth-Fallback-Mode: local_snapshot`	`test_auth_uses_local_snapshot_when_redis_fails`
Redis 异常	当前 worker 无本地快照	JWT-only 短暂放行，并标记 `X-Auth-Fallback-Mode: jwt_only`	`test_auth_uses_jwt_only_fallback_when_redis_fails_and_local_snapshot_missing`
JWT 过期 / 无效	不适用	前端清 token 并回登录页	`tests/test_ui_smoke.py` 4 passed

多 Worker 下，本地缓存是进程私有的。

场景：

Worker 1 缓存了用户 A
Redis 宕机
用户 A 下一次请求打到 Worker 2
Worker 2 没有本地缓存

如果不做 JWT-only fallback，用户会出现“刷新一下能进，再刷新又不能进”的诡异体验。

内部知识库场景下，Redis 已经宕机属于重大故障，此时优先保可用。

内存保护

本地鉴权缓存不是无限 dict，而是 TTL + LRU：

```
AUTH_LOCAL_FALLBACK_TTL_SECONDS=30
AUTH_LOCAL_FALLBACK_MAXSIZE=500
```

最多缓存 500 个用户，过期或最久未访问的会淘汰，避免 Worker 内存无限上涨。

新手解释

JWT 像门票，Redis 用户快照像门口名单。正常时既看门票也看名单；Redis 挂了时，短时间只认门票，避免整个场馆都进不去。

6. ASK 缓存：重复问题不要重复跑 RAG 和 LLM

防止什么问题

同一个问题如果每次都重新：

```
Embedding -> Qdrant -> LLM -> JSON 解析 -> 落库
```

会浪费大量时间和模型费用。

做了什么动作

实现了 ASK Redis 缓存。

缓存里保存的是“标准化后的答案结果”，不保存单次请求的 `request_id`。

因为：

- 每次请求都应该有新的 `request_id`。
- 即使命中缓存，也要写本次历史和审计。

精确缓存

完全相同问题、相同部门、相同模型、相同知识库 collection、相同配置指纹，才会命中。

适合：

- “公司报销流程是什么？”
- 用户反复刷新
- 前端重试

语义缓存

后来增加了语义缓存：

```bash
ASK_SEMANTIC_CACHE_ENABLED=true
ASK_SEMANTIC_CACHE_THRESHOLD=0.97
ASK_SEMANTIC_CACHE_MAX_CANDIDATES=200
```

意思是：

- 问法稍微不一样，但语义非常接近，也可以复用答案。
- 阈值设得很高，降低误命中。

例如：

> 公司报销流程是什么？
> 公司的报销流程是什么？

可能命中。

但：

> 公司报销流程是什么？
> 年假怎么申请？

不会命中。

项目	优化前 / 历史表现	优化后 / 本轮表现	结论
精确同问	第一次约 28.6s，第二次约 24ms	本轮 40/40 cache hit	精确缓存可以稳定复用答案
cache hit 后 retrieve	原链路会进入检索	p50=0ms，p95=0ms，max=0ms	命中后完全绕过 Qdrant 检索
cache hit 后 answer	原链路会调用 LLM	p50=0ms，p95=0ms，max=0ms	命中后完全绕过 LLM 生成
cache hit 总耗时	历史第二次约 24ms	p50 ≈ 184ms，p95 ≈ 440ms	当前本轮高并发口径下仍有 wall 长尾，需要看持久化和 Redis 毛刺
persist_bundle	原来同步写库可能拖主链路	p50 ≈ 19ms，p95 ≈ 306ms	需要确认这是入队耗时还是实际写库仍在主路径

场景	wall	cache_lookup	retrieve	answer	persist_bundle	结论
/ask cache miss，3 并发，3 请求	p50≈1669ms，p95≈5741ms	p50≈0ms，p95≈1ms	p50≈481ms p95≈4297ms	p50≈1168ms p95≈1414ms	p50≈19ms p95≈20ms	miss 长尾主要在 retrieve

能力	当前状态
精确缓存	已实现，非流式和流式都复用
语义缓存	已实现，高相似问题可命中
低相似度保护	已实现，不相干问题不命中
语义 miss 向量复用	已实现，避免重复 embedding

关键优化：避免重复算 Embedding

语义缓存 miss 时，系统已经算过一次问题向量。
如果再进入检索阶段重新算一次，就是浪费。
所以现在逻辑是：

> 语义缓存查找时算 query_vector

-> miss
-> 把 query_vector 传给 retrieve
-> retrieve 不再重复 encode

这能减少 300ms 到 500ms 左右的隐藏浪费，尤其在 CPU 模型上明显。

出问题时会怎么处理

Redis 缓存读写失败：

- 不影响主链路。
- 继续走检索和 LLM。
- meta 里记录 `cache_status=error` 或 `store_failed`。

新手解释

缓存像把常见问题答案贴在前台。有人问一样的问题，前台直接回答；贴纸掉了也没关系，照样去后厨问专家。

---

7. 启动预热：把第一次慢挪到启动时

指标	冷启动观测	热态观测	结论
retrieve latency	约 4527ms	约 194ms - 218ms	冷启动检索慢很多，预热有价值
TTFT	约 2344ms	约 832ms - 875ms	热态首 token 明显更快
用户体验	第一问等待感强	后续问答更流畅	启动预热、模型预加载、连接预热值得保留

防止什么问题

Embedding 模型第一次加载很慢。

如果不预热，第一个真实用户会遇到：

第一次请求 retrieve 4s+
第二次请求 retrieve 200ms

这不是缓存命中，而是模型冷启动。

做了什么动作

API startup 时执行：

加载 embedding 模型
做一次 tiny encode

配置：

RETRIEVAL_WARMUP_ON_STARTUP=true

出问题时会怎么处理

预热失败不会阻止服务启动，只会记录日志。

新手解释

就像餐厅开门前先把炉子烧热。第一位客人不用等厨师点火。

8. 流式响应与 TTFT：让用户先看到字

防止什么问题

普通接口要等完整答案生成完才返回。LLM 生成可能需要几秒，用户会以为系统卡死。

做了什么动作

实现 `/ask/stream` SSE 流式接口：

```
event: status       started
event: status       retrieve_done
event: status       first_token
event: token        ...
event: final        完整结构化答案
event: done         ok
```

并记录：

ttft_ms = Time To First Token

指标	本轮观测值	含义
测试接口	`/ask/stream`	SSE 流式问答接口
测试问题	“公司报销流程是什么？”	热态问题
`retrieve_done latency`	≈194ms	检索完成耗时
`first_token TTFT`	≈875ms	用户看到第一个 token 的时间
`final.meta.latency_ms.answer`	≈2566ms	完整回答生成耗时
`stream_tokens`	210	流式输出 token 数

出问题时会怎么处理

如果模型正常：

- 用户会逐 token 看到输出。
- final 里有完整答案、引用和 meta。

如果模型失败：

- 返回安全拒答。
- `attempt_stage` 会标记失败阶段。

新手解释

流式响应就像客服边想边打字。即使完整回答还没写完，用户也知道系统活着。

9. 异步落库队列：别让审计写库拖慢用户

防止什么问题

ASK 回答已经生成完了，但还要写：

- `kb_queries`
- `audit_logs`

这类审计写库不应该阻塞用户等答案。

压测里 `persist_bundle` P95 可能有几十毫秒。单次不大，但高并发下会占用数据库连接池。

做了什么动作
实现本地异步批量落库队列：
- `src/api/ask_persist_queue.py`
- `asyncio.Queue`
- 定时批量刷库
- shutdown 时最多等待 2 秒 flush
配置：
ASK_ASYNC_PERSIST_ENABLED=true
ASK_ASYNC_PERSIST_QUEUE_MAXSIZE=1000
ASK_ASYNC_PERSIST_BATCH_SIZE=50
ASK_ASYNC_PERSIST_FLUSH_INTERVAL_SECONDS=1.0
ASK_ASYNC_PERSIST_SHUTDOWN_FLUSH_TIMEOUT_SECONDS=2.0

出问题时会怎么处理
- 队列正常：请求只入队，很快返回。
- 队列满：退回同步写库，避免丢数据。
- 服务优雅关闭：尽量 flush 剩余队列。
- `kill -9` 或断电：最多丢最近约 1 秒的审计数据。

写路径场景	优化前	优化后	结论
create 600/60	0 成功，600 失败，p95≈15064ms	593 成功，7 失败，p95≈927ms	建单从完全雪崩变成基本可用
escalate 500/50	500 成功，0 失败，p95≈951ms	500 成功，0 失败，p95≈1034ms	50 并发仍稳定，延迟略升
escalate 600/60	17 成功，583 失败，p50≈15029ms	600 成功，0 失败，p95≈1238ms	60 并发热点更新已稳定
escalate_many 600/60/20	58 成功，542 失败，p95≈15023ms	600 成功，0 失败，p95≈502ms	分散写改善最明显

新手解释

用户吃完饭不用等服务员把账本写完。服务员先让用户走，账本稍后批量补。

10. LLM 熔断：模型挂了，不要所有请求一起傻等
防止什么问题

LLM 供应商可能：
- 超时
- 限流
- API key 错
- 网络抖动
如果每个请求都等 30 秒超时，高并发下 Worker 会被拖死。
做了什么动作
增加：
OPENAI_TIMEOUT_SECONDS=30
OPENAI_STREAM_TIMEOUT_SECONDS=30
OPENAI_CIRCUIT_BREAKER_ENABLED=true
OPENAI_CIRCUIT_REDIS_ENABLED=true

OPENAI_CIRCUIT_FAIL_THRESHOLD=3
OPENAI_CIRCUIT_OPEN_SECONDS=30

逻辑：

LLM 连续失败 3 次
-> 打开熔断 30 秒
-> 后续请求直接返回“模型服务暂时不可用”
-> 不再继续打 LLM

多 Worker 怎么办

熔断状态优先写 Redis：

circuit:llm:failures
circuit:llm:open_until

任何一个 Worker 发现 LLM 挂了，其他 Worker 也会知道。

如果 Redis 自己也不可用，则退回进程内本地熔断。

出问题时会怎么处理

返回安全答案：

证据不足：模型服务暂时不可用，请稍后重试。

meta 里会看到：

attempt_stage = llm_circuit_open
failure_reason = llm_circuit_open:retry_after=...

项目	观测 / 验证结果	结论
真实 LLM 并发 1	墙钟约 1s	单请求基线
真实 LLM 并发 5	墙钟约 3.7s，效率比 0.9x	LLM 请求有重叠，不是完全串行卡死
真实 LLM 并发 10	墙钟约 5.7s，效率比 1.5x	LLM 不是主要 event loop 阻塞点
/ask/stream 热态 TTFT	≈875ms	首 token 指标可观测
LLM 熔断打开	服务测试覆盖 Redis open state 下不调用 LLM	熔断打开后能快速降级，避免继续等待超时

新手解释

熔断像电闸。电器短路时，不要让所有电线继续烧，先断开一会儿。

---

11. Qdrant 熔断：向量库挂了，RAG 要安全拒答

防止什么问题

验证项	预期	当前结果	判断
Qdrant 连续失败单测	熔断打开后快速返回空证据	test_qdrant_circuit_opens_after_dense_failures 通过	熔断逻辑单测通过
多 worker 共享熔断状态	Worker A 打开熔断，Worker B 也能感知	test_qdrant_circuit_honors_redis_open_state 通过	Redis 共享 open state 逻辑通过
docker	返回 200 降级答案或安全	本轮返回 500，wall≈198ms	真实停机路径未通过，必须

compose stop qdrant 真实 chaos	拒答		修复

Qdrant 是向量检索服务。如果它挂了，RAG 找不到证据。

危险情况是：

- 请求一直等 Qdrant 超时。
- Worker 被占满。
- 最后接口 500。

做了什么动作

增加：

```
QDRANT_TIMEOUT_SECONDS=5
QDRANT_CIRCUIT_BREAKER_ENABLED=true
QDRANT_CIRCUIT_REDIS_ENABLED=true
QDRANT_CIRCUIT_FAIL_THRESHOLD=3
QDRANT_CIRCUIT_OPEN_SECONDS=30
```

逻辑：

```
Qdrant 连续失败 3 次
  -> 打开熔断
  -> 后续请求直接返回空证据
  -> 回答层返回“证据不足”
```

出问题时会怎么处理

不会乱编答案。

会返回类似：

证据不足：未检索到与问题相关的文档片段。

状态码仍可以是 200，因为业务上是“无法基于证据回答”，不是程序崩溃。

模式	GoldDoc R@3	GoldDoc R@5	GoldDoc MRR	Auto APC	Citation Output	Refusal	Retrieve p50	Retrieve p95	结论
dense	86.15%	90.00%	0.7955	40.63%	86.15%	14.62%	211ms	308ms	快，但召回略弱
hybrid_rrf	93.08%	96.15%	0.8574	43.31%	88.46%	13.08%	216ms	339ms	默认推荐，R@5 +6.15pp，p50 只多约 5ms
hybrid_rrf_rerank	96.92%	97.69%	0.9164	47.64%	93.08%	6.92%	11192ms	16168ms	最准但太慢，不适合默认在线链路

新手解释

RAG 的原则是“有证据才回答”。档案室打不开时，宁可说“查不到证据”，也不能凭空编。

12. MCP 熔断与并发闸门：Agent 工具不能无限打

防止什么问题

场景	请求/并发	成功	失败	p95	结论
MCP health 小测	5/2	5	0	≈232ms	MCP 协议层有额外开销
MCP lookup 小测	5/2	5	0	≈528ms	MCP lookup 比直连 API 慢
MCP lookup 大测	1000/100	1000	0	≈4258ms	不报错但延迟高，可能因为 MCP session 初始化形成削峰

Agent 可能调用 MCP 工具，比如：

- 查工单
- 建工单
- 续办草稿
- 取消工单

如果 MCP 服务挂了，或者工具调用太多，Agent 会把后端拖垮。

做了什么动作

MCP 远程调用已有熔断：

```
AGENT_MCP_CIRCUIT_BREAKER_ENABLED=true
AGENT_MCP_CIRCUIT_FAIL_THRESHOLD=3
AGENT_MCP_CIRCUIT_OPEN_SECONDS=30
```

MCP wrapper 还有本地并发闸门：

```
MCP_WRAPPER_MAX_CONCURRENT_CALLS=16
MCP_WRAPPER_CONCURRENCY_WAIT_SECONDS=0.2
```

出问题时会怎么处理

- MCP 远程失败多次：返回 `remote_mcp_circuit_open`。
- 工具并发过高：返回 `rate_limited`，可重试。
- 工具参数非法：返回结构化错误，不执行操作。

新手解释

Agent 像实习生，可以帮你调用工具。但不能让实习生无限按按钮，否则系统会被点爆。

---

13. Redis 的角色：缓存、鉴权、全局熔断共享状态

Redis 在这个系统里做三类事：

1. ASK 问答缓存。
2. 鉴权用户快照。
3. 多 Worker 共享熔断状态。

Redis 挂了会怎样

功能	Redis 挂了的表现	当前验证

ASK 缓存	缓存失败开放，继续走 RAG/LLM	缓存读写失败不影响主链路，meta 可记录 cache_status=error 或 store_failed
鉴权快照	先本地短 TTL，再 JWT-only fallback	local snapshot 和 JWT-only fallback 单测已覆盖
LLM/Qdrant 全局熔断	退回本地进程熔断	单测覆盖 Redis open state；Redis 不可用时回退本地状态
限流/幂等	可能返回存储不可用或降级，按具体业务处理	需要按接口策略分别验证

新手解释

Redis 很重要，但不能让 Redis 变成“单点死亡按钮”。所以关键路径都做了失败开放或本地兜底。

14. GIL 与 Python 并发：为什么不是所有慢都能靠 async 解决

什么是 GIL

GIL 是 CPython 的全局解释器锁。简单说：

同一个 Python 进程里，同一时刻通常只有一个线程能执行 Python 字节码。

对这个项目有什么影响

高并发下，即使使用线程池，下面这些纯 Python 工作仍然会抢 GIL：

- JWT 解码
- Pydantic/JSON 序列化
- ORM 对象构建
- 部分业务逻辑

所以：

- `to_thread` 对 I/O 和 torch 这类释放 GIL 的计算有用。
- 对纯 Python 计算不一定有用。
- 多 Worker 可以缓解 GIL，但 Worker 不是越多越好。

实验	优化前 / 观测数据	优化后 / 对照数据	结论
CrossEncoder 直接在 event loop 跑	rerank≈19756ms，心跳最大≈19805ms	—	会把 event loop 冻约 20 秒
CrossEncoder 放进 to_thread	—	rerank≈19304ms，心跳最大≈59ms	总计算时间差不多，但主事件循环不再冻住
缓存命中路径修复前	并发 20 效率比≈11.9x	—	轻路径仍有排队
DB 写 offload 后	—	并发 20 效率比≈9.7x	有改善，但瓶颈没有完全消失
/health 扛 30 并发缓存 ASK	—	p50≈10ms，p95≈11ms，max≈16ms	event loop 没被整体冻住
单 worker vs 4 worker	单 worker 效率比≈11.2x，墙钟≈0.24s	4 worker 效率比≈7.4x，墙钟≈0.14s	多 worker 有帮助，但不是越多越好

新手解释

线程像多个厨师，但 GIL 像厨房只有一把刀。多个厨师抢一把刀，不能无限变快。

15. 当前推荐生产启动方式

Docker Compose 默认建议：

```
docker compose up -d --build --force-recreate api mcp kb-init
```

确认 Worker：

```
docker compose exec api sh -c "ps -ef | grep '[u]vicorn'"
```

确认关键环境变量：

```
docker compose exec api sh -c "env | grep -E 'API_WORKERS|DB_POOL|DB_MAX|REDIS_MAX|ASK_|QDRANT_|OPENAI_.*CIRCUIT|AUTH_' | sort"
```

流式测试：

```
curl -N -X POST http://localhost:8080/ask/stream \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -H "Accept: text/event-stream" \
  -d '{"question":"公司报销流程是什么？"}'
```

16. 故障演练清单

这一步叫 Chaos Testing，意思是主动把依赖停掉，看系统是否优雅降级。

Redis 宕机

docker compose stop redis

预期：

- 已登录用户仍可能通过本地快照或 JWT-only fallback。
- 响应头出现：

X-Auth-Fallback: true

恢复：

```
docker compose start redis
```

Qdrant 宕机

```
docker compose stop qdrant
```

预期：

- `/ask` 不应卡死。
- 返回证据不足类答案。
- 连续失败后 Qdrant 熔断打开。

恢复：

```
docker compose start qdrant
```

LLM API 故障

临时改错 API key 或 base URL，然后连发请求。

预期：

- 前几次可能等待 timeout。
- 达到阈值后，后续请求快速返回：

attempt_stage = llm_circuit_open

PostgreSQL 审计压力

如果开启异步落库：

ASK_ASYNC_PERSIST_ENABLED=true

预期：

- 正常情况下 ASK 请求不等待审计写库完成。

- 队列满时退回同步写库，避免丢数据。
- 优雅停机时尽量 flush。

注意：如果整个 PostgreSQL 停掉，依赖数据库的接口仍可能失败；异步落库主要解决“审计写慢拖慢请求”，不是让所有 DB 业务在数据库宕机时照常运行。

方向	证据	结论
建单写路径	create 600/60：0 成功 -> 593 成功，p95≈15064ms -> p95≈927ms	从完全雪崩变成基本可用
单工单催办	escalate 600/60：17 成功 -> 600 成功	原 60 并发雪崩点已越过
多工单分散更新	escalate_many 600/60/20：58 成功 -> 600 成功，p95≈15023ms -> p95≈502ms	分散写路径改善最明显
RAG 检索策略	dense R@5=90.00%，hybrid_rrf R@5=96.15%；p50 211ms -> 216ms	hybrid_rrf 是默认在线检索的合理选择
ASK 缓存	cache hit 后 retrieve=0ms，answer=0ms	缓存命中确实绕过检索和 LLM
流式响应	热态 retrieve≈194ms，TTFT≈875ms，完整 answer≈2566ms	用户可以先看到 token
Event loop 优化	CrossEncoder 直接跑心跳最大≈19805ms，to_thread 后≈59ms	CPU 重活不能堵主事件循环

17. 常见误区

误区 1：并发数等于连接数

`wrk -c 30` 表示 30 个连接，不一定等于 30 个真实用户业务请求。

如果真实环境是短连接，要注意 keepalive 行为。

误区 2：Debug 模式也能压测

不要用：

--reload

--log-level debug

压测要用生产配置，否则结果会失真。

误区 3：缓存越激进越好

语义缓存阈值不能太低。

如果把阈值设成 0.8，可能把“报销流程”和“年假申请”误判成相似，返回错误答案。

误区 4：熔断就是报错

熔断不是为了报错，而是为了“快速失败、保护系统”。

LLM 挂了时，不要所有请求都等 30 秒。直接告诉用户“模型暂不可用”，系统才能继续活着。

误区 5：本地缓存等于全局缓存

多 Worker 下，每个 Worker 是不同进程。

本地内存：

Worker 1 有

Worker 2 不一定有

所以全局状态要放 Redis，比如 LLM/Qdrant 熔断状态。

18. 这轮优化的价值总结

优化	防住的问题	故障时表现
请求打点	不知道慢在哪里	响应 meta/log 能看到分段耗时
固定 Worker/连接池	Worker 太多、连接爆炸	资源上限可控
Redis 鉴权快照	每次请求查用户表	热路径不查 DB
本地/JWT 鉴权兜底	Redis 宕机导致全员掉线	短时间放行并打响应头
ASK 精确缓存	重复问题重复调用 RAG/LLM	中后直接返回缓存答案
ASK 语义缓存	相似问法重复生成	高相似度复用答案
query_vector 复用	语义 miss 后重复算 embedding	少算一次 embedding
启动预热	首问冷启动慢	启动时提前加载模型
流式响应	用户等待完整答案	先看到 token
TTFT 记录	不知道首字慢在哪里	可区分检索慢还是模型慢
异步落库	审计写库拖慢响应	入队后返回，后台批量写
LLM 熔断	模型挂了拖死 Worker	快速返回模型暂不可用
Qdrant 熔断	向量库挂了请求卡死	返回证据不足
MCP 熔断/并发闸门	工具调用打爆后端	限流或短期开路

19. 当前系统还不是“无限可靠”

已经做到的是：

- 高并发下更可观测。
- 常见外部依赖故障可降级。
- 多 Worker 下熔断状态可共享。
- 鉴权 Redis 故障时用户体验更稳定。

仍然要知道边界：

- JWT-only fallback 是可用性优先，安全性会短时间放宽。
- 本地异步落库在 `kill -9` 或断电时可能丢最近约 1 秒审计。
- PostgreSQL 整体宕机时，依赖数据库的业务接口仍会受影响。
- Redis 如果长时间宕机，缓存、全局熔断、限流、幂等等能力都会退化。
- 语义缓存阈值要保守，不能为了命中率牺牲正确性。

20. 最后给未来的我

这轮优化真正重要的不是某个参数，而是思路：

先观测，再定位；
先底层，再全链路；
先保守缓存，再语义缓存；
先 timeout，再熔断；
先失败开放，再记录可观测标记；
先单机够用，再考虑分布式复杂方案。

系统优化不是一次做完的，它像体检。
这份文档以后可以作为“体检报告”来用：系统慢了，就回到打点；系统崩了，就看是哪一层兜底没生效；要扩容了，就先算 Worker、连接池和共享依赖。