当然可以。下面给你一份**“可拔插更换手册”：针对五个模块（Graph / Retrieval / Reasoning / Verification / Telemetry）+ LLMs Router**，分别说明要点、最小实现骨架、如何在 `settings.yaml` 里切换、以及常见坑。全部基于你现有的抽象接口与装配逻辑，因此替换是**零侵入**的。

---

1. 总开关（你已经具备）

- **抽象接口**：`app/core/interfaces.py`（四大业务）与 `app/telemetry/sinks.py`（TelemetrySink）。
- **DTO 契约**：`app/core/dto.py`（跨模块 I/O 统一）。
- **装配**：`app/di/factory.py`（已兼容 *字符串* 和 *字典* 两种模块配置）。
- **编排**：`app/orchestrator/*` 只依赖接口，不依赖实现类。

所以：**实现一个符合 Protocol 的类 → 在 `settings.yaml` 把模块路径换成你的类即可**（无须改 Orchestrator）。

---

1. 1) Graph Construction（图构建模块）

2. 接口约定

- **Protocol**：`GraphConstruction.build(in: GraphBuildIn) -> GraphBuildOut`
- **输入要点**：`trace_id, question_text, context(list|[(title,[sents])]), graph_id? nodes? edges?`
- **输出要点**：`graph_id, node_count, edge_count, [nodes], [edges], diagnostics, provenance, extra`

3. 最小实现骨架

```
# app/modules/graph_construction/impl_my_gc.py
from app.core.interfaces import GraphConstruction
from app.core.dto import GraphBuildIn, GraphBuildOut

class MyGraphConstruction(GraphConstruction):
    def __init__(self, **kwargs):
        # 可注入 root_dir / 索引等
        self.cfg = kwargs

    def build(self, req: GraphBuildIn) -> GraphBuildOut:
        # 你可以自己生成 nodes/edges，或复用 req.nodes/req.edges
        nodes = req.nodes or []
        edges = req.edges or []
        # ... 组装/落盘（可选）
        return GraphBuildOut(
            graph_id=req.graph_id or f"graph-{req.trace_id}",
            node_count=len(nodes),
```

```
            edge_count=len(edges),
            nodes=nodes, edges=edges,
            diagnostics={"impl": "MyGraphConstruction"},
            provenance={"source":"my_gc"},
            extra={}
        )
```

4. 如何切换

```
# config/settings.yaml
modules:
  graph_construction:
"app.modules.graph_construction.impl_my_gc:MyGraphConstruction"
  # 或
  # graph_construction:
  #   impl: "app.modules.graph_construction.impl_my_gc:MyGraphConstruction"
  #   kwargs: { root_dir: "data/graph" }
```

5. 常见坑

- `GraphBuildOut` 必须返回**计数**正确，否则下游统计会偏差。
- 若持久化，建议把路径写入 `extra.paths`（便于可视化/调试）。

---

6. 2) Retrieval Agent（检索模块）

7. 接口约定

- **Protocol**：`RetrievalAgent.retrieve(in: RetrievalIn) -> RetrievalOut`
- **输入要点**：`query, graph_id, top_k, trace_id`
- **输出要点**：`hits: List[Hit(id, score, meta?)], diagnostics`

8. 最小实现骨架（例如 BM25 + 图邻域）

```
# app/modules/retrieval/impl_bm25.py
from typing import List
from app.core.interfaces import RetrievalAgent
from app.core.dto import RetrievalIn, RetrievalOut, Hit

class BM25Retriever(RetrievalAgent):
    def __init__(self, index_path: str = "data/hotpotqa/docs.jsonl", **_):
        # 加载/初始化你的索引
        self.index_path = index_path

    def retrieve(self, req: RetrievalIn) -> RetrievalOut:
        # 1) 文本检索（示意）
        text_hits: List[Hit] = [Hit(id="Doc#A", score=0.91)]
        # 2) 图扩展（可选，基于 req.graph_id）
        graph_hits: List[Hit] = [Hit(id="Node:X", score=0.80)]
        # 3) 融合与截断
```

```
        hits = sorted(text_hits + graph_hits, key=lambda h: h.score,
reverse=True)[: (req.top_k or 20)]
        return RetrievalOut(hits=hits, diagnostics={"index":
self.index_path})
```

9. 如何切换

```
modules:
  retrieval:
    impl: "app.modules.retrieval.impl_bm25:BM25Retriever"
    kwargs: { index_path: "data/hotpotqa/docs.jsonl" }
```

10. 常见坑

- `Hit.id` 要稳定可追踪（你后面可能把它当作证据来源）。
- 若用 LLM 做 query-expansion，务必**通过 Router** 调用（Orchestrator 已把 `trace_id` 传进 `require`）。

---

11. 3) Reasoning Agent（推理模块）

12. 接口约定

- **Protocol**：`ReasoningAgent.reason(in: ReasoningIn) -> ReasoningOut`
- **输入要点**：`question, hits, graph_id, trace_id`
- **输出要点**：`answer, evidence_used(hits), steps[], model`

13. 最小实现骨架（例如 ReAct 策略）

```
# app/modules/reasoning/impl_react.py
from app.core.interfaces import ReasoningAgent
from app.core.dto import ReasoningIn, ReasoningOut
from app.core.llm_router import LLMRouter

class ReActReasoner(ReasoningAgent):
    def __init__(self, router: LLMRouter, **_):
        self.router = router

    def reason(self, req: ReasoningIn) -> ReasoningOut:
        plan = self.router.complete(
            module="ReasoningAgent", purpose="plan",
            prompt=f"Decompose:\nQ: {req.question}",
            require={"trace_id": req.trace_id}
        )["text"]
        synth = self.router.complete(
            module="ReasoningAgent", purpose="synthesize",
            prompt=f"Use evidence {req.hits} to answer:\nQ:
{req.question}\nPlan:\n{plan}",
            require={"trace_id": req.trace_id}
        )
        return ReasoningOut(
            answer=synth["text"], evidence_used=req.hits,
            steps=[{"plan": plan}], model=synth.get("_model")
```

```
        )
```

14. 如何切换

```
modules:
  reasoning: "app.modules.reasoning.impl_react:ReActReasoner"
```

15. 常见坑

- **必须**把 `req.hits`（你真正用到的证据）放回 `evidence_used`，便于验证与可解释性。
- 保留 `steps`（哪怕只是一段 plan）以便可视化。

---

16. 4) Verifier Agent（验证模块）

17. 接口约定

- **Protocol**：`VerifierAgent.verify(in: VerifyIn) -> VerifyOut`
- **输入要点**：`answer, evidence(hits), graph_id, trace_id`
- **输出要点**：`status("passed"/"warn"/"fail"), findings[], model?`

18. 最小实现骨架（规则 + LLM 一致性）

```
# app/modules/verification/impl_claimcheck.py
from app.core.interfaces import VerifierAgent
from app.core.dto import VerifyIn, VerifyOut
from app.core.llm_router import LLMRouter

class ClaimCheckVerifier(VerifierAgent):
    def __init__(self, router: LLMRouter, **_):
        self.router = router

    def verify(self, req: VerifyIn) -> VerifyOut:
        rule_findings = [] if req.evidence else [{"issue": "no_evidence"}]
        llm = self.router.complete(
            module="VerifierAgent", purpose="factcheck",
            prompt=f"Check consistency:\nAnswer: {req.answer}\nEvidence:
{req.evidence}",
            require={"trace_id": req.trace_id}
        )["text"]
        status = "passed" if not rule_findings else "warn"
        return VerifyOut(status=status, findings=rule_findings + [{"llm":
llm}], model=None)
```

19. 如何切换

```
modules:
  verification:
"app.modules.verification.impl_claimcheck:ClaimCheckVerifier"
```

20. 常见坑

- `status` 的语义要稳定（推荐三态：passed/warn/fail）。
- `findings` 里给出可读、可追踪的条目（便于 UI / 日志端呈现）。

---

21. 5) Telemetry（日志采集/可视化模块）

22. 接口约定

- **Protocol**：`TelemetrySink.record(event)`, `flush_run(trace_id, result)`
- 调用点：Orchestrator 节点（`span(...)`），Router（`record_llm_call(...)`）。

23. 最小实现骨架（将日志打到 ELK/Kafka 等）

```
# app/telemetry/sink_elastic.py
from typing import Dict, Any
from app.telemetry.sinks import TelemetrySink, TelemetryEvent
import json

class ElasticSink(TelemetrySink):
    def __init__(self, endpoint: str, index: str = "rag_runs", **_):
        self.endpoint = endpoint; self.index = index

    def record(self, evt: TelemetryEvent) -> None:
        # 伪代码：POST 到你的日志系统
        # requests.post(f"{self.endpoint}/{self.index}/_doc", json=evt)
        pass

    def flush_run(self, trace_id: str, result: Dict[str, Any]) -> None:
        # 写入最终快照
        # requests.post(..., json={"trace_id": trace_id, "result": result})
        pass
```

24. 如何切换

目前 `system.py` 默认创建 `LocalJsonlSink`。你可以改为从 `settings.yaml` 读取：

```
# app/system.py (示意片段)
from app.di.factory import import_from_string, load_settings
def init_system():
    settings = load_settings("config/settings.yaml")
    sink_cfg = settings.get("telemetry", {"impl":
"app.telemetry.sinks:LocalJsonlSink", "kwargs": {"root_dir":"runs"}})
    SinkCls = import_from_string(sink_cfg["impl"])
    sink = SinkCls(**(sink_cfg.get("kwargs") or {}))
    ...

settings.yaml:

telemetry:
```

```
  impl: "app.telemetry.sink_elastic:ElasticSink"
  kwargs: { endpoint: "http://localhost:9200", index: "rag_runs" }
```

25. 常见坑

- 确保 **run_start / node_start / llm_call / metrics / run_end** 事件都能被接收并可查询。
- 若替换为异步/网络 Sink，注意失败重试与缓冲。

---

26. 6) LLMs Router（不算“模块”，但最常被自定义）

27. 如何替换 Provider

实现 `LLMProvider` 并注册到 `providers`：

```
# app/core/providers/my_provider.py
from app.core.providers.base import LLMProvider

class MyProvider(LLMProvider):
    def complete(self, model: str, prompt: str, require: dict) -> str:
        return "my provider output"
    def embed(self, model: str, texts: list[str], require: dict) ->
list[list[float]]:
        return [[0.0]*3 for _ in texts]
```

`settings.yaml`：

```
providers:
  myprov:
    type: "app.core.providers.my_provider:MyProvider"
    kwargs: { }

llm_policy:
  routes:
    ReasoningAgent:
      plan:   [ { provider: "myprov", model: "my-model" } ]
      synthesize: [ { provider: "myprov", model: "my-model" } ]
```

Router 不需要改；通过 `llm_policy.routes` 就能把**任意模块/目的**路由到你的 Provider/Model。

---

2. 更换流程 checklist（通用）

1. **放置你的实现类**（推荐放在对应目录下，例如 `app/modules/retrieval/impl_xxx.py`）。
2. **实现对应 Protocol 方法**（签名与 DTO 一致）。

3. （可选）实现 `@classmethod from_settings(cls, settings, router)` 用于复杂构造。
4. **在 `settings.yaml` 指定实现路径**（字符串或字典带 kwargs；你已兼容两者）。
5. （可选）若你的实现内会调 LLM：通过 **Router** 调用，并把 `trace_id` 传到 `require`，以保证 LLM 调用也进入日志。
6. 跑 `my_code/run_system.py` 验证；看 `runs/<trace>/events.jsonl` 与 `run.json`。

需要我根据你现在的 `settings.yaml` 直接给出**替换成你自定义模块**的具体片段吗？或者挑一个模块（比如 Retrieval）我给你一份**完整可跑的类 + 改完的 settings**。