ChatGPT

Unreal 引擎项目“记忆系统”数据基础设施方案

本文构思了一套围绕 Unreal 引擎项目的“记忆系统”数据基础设施方案，旨在采集、存储、结构化并回溯项目运行、优化和迭代过程中的数据，为未来构建基于类 Claude Code 的 AI Agent 提供可学习、可检索的知识库支持。方案涵盖数据来源与采集方式、数据标准化与模式、数据存储策略、自动化管线设计、AI Agent 接口规划以及 MVP 路线图。

数据来源与采集方式

为了全面记录 Unreal 项目的运行与优化过程，需要从多个来源采集数据。主要的数据源及其采集方式包括：

- **Unreal Insights Trace（跟踪数据）**：利用 Unreal Insights 的追踪功能高频捕获引擎事件和性能数据[1]。Unreal Insights 是虚幻引擎的遥测分析套件，可在项目运行时记录各线程的事件、函数调用、帧时间、内存/CPU/GPU 使用等。[1] 采集方式：在运行游戏时启用 Trace（例如使用命令行参数如 `-trace=cpu,gpu,frame,log`），由 Unreal Trace Server 后台记录 `.utrace` 会话文件[2]。每次运行或测试场景结束后，将生成的 `.utrace` 文件和相关 `.ucache` 数据保存下来，作为后续分析的原始数据输入。

- **日志（Log）**：Unreal 引擎运行时产生的日志文本（包含调试信息、警告、错误等）。采集方式：配置引擎将日志输出到文件（位于 `Saved/Logs` 目录）或通过插件将日志流发送到集中服务器。日志包含时间戳、日志级别、模块来源和消息内容等信息。应确保在**开发版**或**测试版**开启足够的日志级别，以获取详细信息。对于关键事件，可在代码或蓝图中插入自定义日志或书签，辅助定位问题（例如使用 `UE_LOG` 或 Blueprint 的打印节点）。

- **Stat 性能统计**：Unreal 内置的 `stat` 命令提供帧率、线程用时、内存占用等实时指标[3][4]。采集方式：通过控制台批量执行相关 `stat` 命令（如 `stat unit`、`stat memory`）定期采样数据，或者利用引擎提供的 Profiling API 自动记录这些统计值。许多 stat 指标也可通过 Insights trace 捕获（例如启用 `cpu,gpu,frame` trace 会话已包含帧耗时等数据），因此可以将 stat 数据作为 Trace 事件的一部分统一采集。

- **配置和环境（Config）**：项目的配置文件（如*.ini、项目设置）以及运行环境参数（引擎版本、平台信息、使用的插件版本等）。采集方式：在每次运行或版本迭代时，记录关键配置项的值和变更。可以编写脚本将 `.ini` 配置解析成结构化数据，或者利用引擎的配置读取接口导出 JSON。环境信息如引擎版本、操作系统、硬件规格等也要一同记录，便于关联性能数据与环境。

- **蓝图和关卡结构（Blueprint & Asset Structure）**：项目内容的静态结构数据，如蓝图脚本的节点和连线、关卡场景中Actor和组件的数量，以及代码版本仓库信息等。采集方式：在编辑器阶段使用反射或编辑器插件导出蓝图/关卡信息。例如，可编写Editor Utility，将特定蓝图的节点（类型、复杂度）导出成JSON，或利用资产注册表查询关卡中Actor数量等。这样的**工件数据**有助于分析哪些系统可能导致性能瓶颈（如蓝图复杂度是否与帧率相关）。

上述多种数据源通过**引擎插件或自动化脚本**进行采集：在项目运行时启动Trace记录和日志监听，在运行结束后收集生成的文件和数据。在移动端或主机平台上运行时，可通过网络将Trace和日志上传到PC上的Trace Server或日志收集服务，实现跨平台的数据获取。由于Unreal Insights的Trace会话具备自描述性，并与不同引擎版本兼容[2]（即使引擎升级，旧的trace数据仍可解析），该方案在跨版本演进时具有良好的可复现性和向后兼容能力。

数据标准化（事件、工件、总结）与 Schema 示例

为了让后续AI Agent能够理解和检索，需将采集的多源数据规范化为统一的结构。我们将数据分为三类：**事件**、**工件**和**总结**，并为每类定义清晰的Schema：

- **事件 (Event)**：指在系统运行过程中发生的动态事件或时序记录。例如日志条目、Trace采样、性能计数、错误崩溃等。事件应包含**时间戳**、**类型**、**来源**、**内容**等字段，以及必要的上下文。
  示例 *Schema*（JSON 表示）：

```
{
  "event_id": "EVT-20231101-142355-001",
  "timestamp": "2023-11-01T14:23:55.123Z",
  "event_type": "LOG",
  "subtype": "Warning",
  "source": "PhysicsSubsystem",
  "message": "Physics threshold exceeded for Actor X",
  "severity": "Warning",
  "session_id": "Session-20231101-1",
  "engine_version": "5.2.1",
  "platform": "Windows"
}
```

  上例表示某次运行中的一条日志警告事件。`event_type`可取值如`LOG`、`TRACE`、`STAT`、`ERROR`等，`subtype`细化类别或频道（例如日志级别、trace通道名称）。通过统一的事件Schema，我们可以将不同来源的数据纳入同一时间轴，并记录其元数据（如引擎版本、平台环境），实现上下文关联和跨平台比较[5]。特别地，对于Trace采集的事件，可以按引擎模块或线程归类（如"渲染线程/Frame事件"、"GC垃圾回收事件"等），字段中加入`thread`、`category`等标签。**每个事件尽可能附带来源引用**（如脚本文件名、蓝图名、Actor名）以便关联到静态工件。

- **工件 (Artifact)**：指相对静态的内容或产出物，包括代码和资产本身，以及由运行过程产生的实体。典型的工件有蓝图脚本、关卡资产、配置文件、二进制构建产物、甚至特定时刻的内存转储或性能报告文件等。工件数据用于提供背景知识，例如某错误日志关联的蓝图脚本内容，或某性能瓶颈涉及的关卡场景复杂度。
  示例 *Schema*：

```
{
  "artifact_id": "BP_PlayerController_v12",
  "artifact_type": "Blueprint",
  "name": "PlayerController",
  "version": 12,
```

```
  "data": "（此处可存储蓝图结构的序列化JSON，或是指向对象存储的引用）",
  "last_modified": "2023-10-20T10:15:00Z",
  "engine_version": "5.2.1",
  "related_assets": ["PlayerPawn", "UI_HUD"]
}
```

上例是一个蓝图资产的工件信息，包含版本号、序列化后的结构数据等。`artifact_type`可能是`Blueprint`、`Level`、`Config`、`LogFile`、`TraceFile`等。对于大型工件（如完整的日志文件、trace文件），`data`字段可存储文件路径或对象存储的引用，而非原始内容。工件应记录其**版本或变更标识**（例如Git提交哈希，蓝图的Revision等），以支持**版本差异**分析。通过规范化工件，Agent 可检索项目的知识**背景**——例如查询某错误相关的配置项当前值，或加载某蓝图的逻辑流程图。

- **总结 (Summary)**：指由事件和工件提炼出的**高层次信息**，包括分析报告、性能摘要、问题根因和解决方案等。这类数据可由自动化工具生成，也可以是开发者人工撰写的笔记或复盘文档（如一次性能优化的经验总结）。总结提供对底层数据的凝练和模式洞察。
  示例 *Schema*：

```
{
  "summary_id": "SUM-20231101-001",
  "summary_type": "PerformanceSummary",
  "scope": "Map01_NightBattle Scene, Session-20231101-1",
  "created_at": "2023-11-01T15:00:00Z",
  "content": "在Map01夜战场景中，本次运行的平均帧时间为18ms，其中GPU耗时约11ms，CPU耗时7ms。检测到主要瓶颈在于光照渲染（占GPU帧时5ms）。建议优化方向：减少动态光源数量或使用烘焙光照。",
  "related_events": ["EVT-20231101-142355-150", "..."],
  "related_artifacts": ["Level_Map01", "ScalabilitySettings"]
}
```

该示例总结了某场景下的一次性能分析结果，包括帧时间和瓶颈原因。`summary_type`可以是`PerformanceSummary`（性能汇总）、`ErrorPostmortem`（错误事后分析）、`VersionDiff`（版本差异报告）等。Summary 的`content`既可以是**自然语言描述**，也可以是结构化的JSON（例如性能指标的表格）。我们鼓励**自动化生成**初步总结，再由开发者校验补充。例如，系统在获取一段冗长Trace后自动生成"瓶颈叙述" [6] [7]，提炼出CPU或GPU的主要开销点，作为PerformanceSummary存储；或比较两次运行的Trace/配置差异生成VersionDiff摘要。总结数据将作为AI Agent **知识检索的核心**，因为它提供了浓缩的信息和专家见解，Agent 可以直接引用或基于此进行推理。

通过上述三类结构，所有数据都可以归一化表示：实时发生的**事件流**提供细粒度事实，静态**工件**提供背景和上下文，而**总结**提供人类可读的高层知识和模式。每条数据都附带元数据（时间、版本、来源等），确保在检索时能够按条件筛选（例如按时间段、特定版本、特定场景） [5]。

数据存储策略

针对不同类型的数据和访问需求，我们设计分层次的存储方案，以在检索性能和存储成本之间取得平衡：

- **关系型数据库（Relational DB）**：用于存储**结构化元数据**和**索引**信息。关系库非常适合存放事件和工件的**索引记录**，便于通过SQL查询筛选。例如，将事件表按`event_id`、`timestamp`建立索引，支持时间范围过滤和条件查询（如按`event_type`筛选错误事件）。工件表则记录各资产的版本、类型等信息，可查询特定版本的配置或蓝图。关系数据库（如MySQL/PostgreSQL）提供事务和复杂查询能力，方便实现关联查询（如JOIN事件与对应工件）。例如，查询"某版本之后发生过哪些崩溃事件，并找出相关的蓝图"可在SQL中通过版本和引用ID关联两类数据完成。

- **对象存储（Object Storage）**：用于保存**大体积、非结构化原始数据**文件。如完整的日志文件、Trace二进制文件(.utrace)、内存转储、蓝图资产文件(.uasset)等。对象存储可以是云存储（如S3）或本地文件系统/网络盘。关系型数据库中仅存这些对象的引用路径或ID，而实际内容以文件形式存放，便于后续详细分析或重放。例如，当AI Agent需要深入分析某次性能测试的全部细节时，可以根据引用取回原始`.utrace`文件，在本地使用 Unreal Insights GUI 打开或用解析库处理。对象存储也适合**版本管理**：可以按运行会话或版本号组织文件路径，实现不同版本的数据隔离又可比对。

- **向量数据库（Vector DB）**：用于存储**语义向量索引**，支撑智能检索和关联推荐。向量库保存从事件、工件和总结中提取的文本向量嵌入（embedding）。当AI Agent以自然语言查询时，可以将查询语句也转换为向量，在该库中进行**语义相似度**检索，找到相关的内容切片。由于日志和总结多为非结构化文本，语义检索对解决"措辞不同但含义相关"的查询特别有效[8][9]。我们考虑为不同数据类型建立**独立的向量索引**空间，以获得更精确的匹配[10][11]。例如，蓝图/代码片段的向量索引可采用专门的代码嵌入模型，日志/事件的索引采用通用文本嵌入模型，而总结文档的索引用通用文档模型。这些索引可以存放于开源或商用的向量数据库中（如 Qdrant、Milvus、Weaviate、Pinecone 等都是低延迟的ANN向量检索方案[12]）。每条向量数据需关联元数据（如对应的原始内容ID、来源类别、时间）以便过滤和结果解释。

- **缓存与搜索引擎**（可选）：为了提升检索灵活性，我们还可引入全文搜索引擎（如 Elasticsearch/OpenSearch）来支持**关键字**检索和**复杂过滤**。例如，对日志和配置内容既可以做向量相似搜索，也可以通过关键字（如特定错误代码）精确匹配[11]。混合检索能够兼顾语义相关性和精确匹配，在调试场景下非常有用。实现上，可以将Elasticsearch与向量数据库结合（有些引擎支持同时存储向量和关键词索引），或由应用层同时查询两者合并结果[13]。缓存方面，可针对经常访问的Summary或最近的Session数据做缓存，加速Agent查询响应。

综上，数据存储分为**热数据**（关系DB索引 + 向量索引，支持快速查询）和 **冷数据**（对象存储原始文件，必要时取用）。所有数据存储都需要包含**版本/环境标签**以支持跨版本分析和**横向对比**（例如可以同时检索多个版本的相同事件类型进行比较）。由于Trace数据量巨大且时间敏感，我们可以制定**分片和归档策略**：例如向量索引中只保留最近N天或N次运行的详细事件的向量，以控制索引规模，其余历史数据的embedding定期转移出热存储，改由冷存储通过关键词检索。[14] 这样的策略保证知识库既新鲜又不膨胀，同时保留长期回溯的可能。

自动化管线设计

要让上述数据采集和存储流程高效运作，需要构建一条端到端的**自动化数据管线**。管线的主要阶段和设计如下：

1. **数据采集与上传**：在每次运行或构建流程中触发数据采集任务。例如，集成到CI流程：运行特定性能测试关卡-> 引擎启动时开启Trace和需要的日志 -> 测试结束退出时自动将日志和Trace文件保存到指定位置。可以开发一个**Unreal引擎插件**或启动脚本，在游戏退出后自动执行上传：将新生成的`.utrace`和日志文件发送到服务器或拷贝到集中存储，并调用接口通知后续处理。对于开发者本地的非自动运行，可提供一个工具脚本，一键执行"运行+采集+上传"过程，减少人工介入。

2. **解析与标准化**：当新的原始数据上传后，触发ETL（Extract-Transform-Load）作业解析文件并标准化为前述Schema。具体包括：解析日志文本（按行生成事件对象，提取时间戳、级别等字段），解析Trace文件（可使用官方提供的解析库或JSON导出工具，将重要事件如帧标记、CPU/GPU采样、Trace Bookmarks转换为事件对象）。同样，对配置文件和蓝图结构进行转换：例如检查本次运行对应的项目Git版本，提取该版本的关键配置和蓝图信息，生成工件对象。如果蓝图自上次版本有更新，需生成新的artifact记录并存档老版本。此步骤可以由后端的**解析服务**完成，针对不同数据源有相应的解析模块。产出的结构化Event列表、Artifact列表存入关系数据库，对应的原始文件存入对象存储。为确保跨平台一致性，解析逻辑需涵盖不同平台日志格式差异（比如Windows vs Linux路径格式），统一字段风格（例如路径统一使用正斜杠，时间戳标准化为UTC）。

3. **摘要生成**：在标准化数据就绪后，自动执行**总结生成**任务。基于收集到的一批事件和工件数据，可以产出多种summary：

4. **性能摘要**：如果trace包含帧性能数据，则运行分析脚本计算平均帧率、各线程开销占比、发现帧时间尖峰（hitch）的位置及原因。例如，定位某帧消耗过高是因为触发了垃圾回收，或某蓝图Tick过慢。将这些发现写入PerformanceSummary文本 [7] 。必要时可调用预先训练的模型或规则引擎来**描述性能瓶颈**（例如根据trace事件模式识别常见瓶颈模式，如"高DrawCall造成渲染线程瓶颈"）。
5. **错误摘要**：扫描日志中的 Error/Warning 事件，如发现崩溃，记录崩溃发生频率、调用栈（若有）等，形成问题概述；或者多次出现的Warning汇总统计出现次数、首次/末次时间。对于新的错误类型，还可以自动生成一个issue条目供团队跟踪。
6. **版本变化摘要**：对比当前运行和上一次运行/版本的工件清单，如配置和蓝图更改，看看性能或错误有何差异。如果当前版本性能退化，summary应指出与前版本相比哪项指标变差，并关联可能的改动（例如"自从修改渲染分辨率为4K，GPU每帧耗时增加了X毫秒"）。
7. **知识文档汇总**：对于定期的阶段（如每周构建）可以产出一份长文档，包括该周期所有重要事件的总结和解决措施。这类似于"运行手册/事后分析"的自动化版本，可供人阅读校验。

摘要生成可通过**模板+脚本**实现固定格式的数据汇总，也可探索使用**LLM**模型辅助生成自然语言描述。在早期MVP中，可从简单规则入手（如帧率低于阈值则记录"帧率问题"），后期随着知识库丰富，可尝试用已有总结训练微调模型来生成更复杂的报告。所有生成的Summary应与关联的事件/工件ID链接，方便追溯细节。

1. **Embedding 向量构建**：在获取结构化数据和摘要后，下一步是为**检索增强**准备向量表示。采用预选的嵌入模型分别处理不同内容：
2. 针对**日志和事件**文本，使用通用的文本嵌入模型将每条事件消息或成段日志转换为向量；也可将相关联的一组事件（例如一次崩溃前后的几十行日志）合并为一个文本片段来嵌入，以提供更多上下文 [15] 。

3. 针对**蓝图/代码**工件，可使用代码语义嵌入模型，将函数或蓝图节点描述转换为向量，捕捉其功能含义和结构。因为代码类文本对精确匹配要求高，我们同时保留函数名/节点名等关键字，方便必要时Agent精确查询[11]。
4. 针对**总结文档**，使用通用文档嵌入模型生成向量，以便语义搜索整篇摘要内容。

嵌入计算完成后，将向量和对应元信息存入向量数据库中，分别维护**事件向量索引**、**工件向量索引**、**摘要向量索引**等。为提高检索准确性，我们可以采用**Hybrid 检索**：向量检索提供语义相关度，高精度场景下再结合关系DB或全文检索校正。例如，Agent 查询包含特定错误代码时，先用关键字锁定相关事件，再对这些事件的嵌入做相似度排序，得到最终结果[13]。这些细化策略在初始阶段可不完全实现，但设计上应考虑接口的灵活性。

1. **数据更新与维护**：管线需要具备持续运行能力，保证知识库及时更新和演进。为此要考虑：
2. **定期运行**：在CI中针对关键场景每日或每周运行数据收集，以不断丰富性能档案和问题库。
3. **版本追踪**：每次数据进入时，记录版本号，若引擎或项目版本升级，可在Summary中自动记录"引擎升级导致性能变化"的观察。由于Trace数据与引擎版本无关且自描述[2]，可直接比较不同版本的trace事件。然而对于日志（其内容可能跨版本有变化）需要额外维护**版本映射**（例如某警告信息在新版本中更名，需要在知识库中做关联）。
4. **数据质量监控**：如果解析失败或数据缺失（例如某次Trace文件损坏或日志不全），系统应有告警，提示开发者重新收集。这可通过管线的日志和校验步骤实现。
5. **隐私和容量**：定期清理或归档过旧的详细数据，或者对敏感信息（如玩家个人数据，如果日志涉及）进行脱敏存储。对于AI Agent非必要用到的底层数据，可以在存储时打标，防止被无用地检索，提高检索效率和安全性。

整个管线设计以自动化、最小人工干预为目标。一旦设置完成，每当开发者进行性能测试、调优或者版本发布时，这套系统都会无缝地捕获过程数据，转化为标准知识，做好随时被AI Agent查询的准备。**实时性**也是考虑因素之一：未来可改进为近实时管线，例如通过消息队列（Kafka等）实时推送日志/事件，再微批量更新索引，使得Agent在故障发生几秒内就能获取最新信息[16]。现阶段MVP则可接受分钟级延迟，先确保可靠性。

面向未来 AI Agent 的接口规划

有了规范的知识库，我们需要设计接口以供未来的 AI Agent 使用这些数据，实现诸如 **RAG（Retrieval-Augmented Generation）**问答、**根因分析推荐**和**模型微调**等目标。

- **检索增强生成接口（RAG Query API）**：提供自然语言查询知识库的能力。可以实现一个服务接口，Agent提供查询（例如"最近一次帧率下降的原因是什么？"），系统将：
- 将查询解析出意图和关键词，例如识别要查询的是性能问题，限定"最近一次"。
- 在向量数据库中检索相关**摘要**和**事件**片段。例如找到最近的PerformanceSummary中关于帧率的条目，或查找事件索引中帧率骤降的事件记录。

- 将结果片段返回给Agent，包括内容文本及其来源引用（例如对应的summary或日志片段ID）。Agent 随后把这些上下文作为提示，利用LLM生成回答并引用来源。这一过程即RAG模式，可有效避免单纯LLM对知识遗漏或幻觉[9]。
  接口需支持**多模态检索**：即针对**事件**、**工件**、**总结**各自检索，再融合结果。比如查询一个错误，既可能检索到日志事件，也可能检索到维基/总结中的已有解决方案。为了提高准确度，可以在接口层加入**筛选参数**（如时间范围、版本号），或者内置一些**重排序逻辑**（如偏重最新数据或高优先级事件）。总之，该查询接口要让Agent灵活地获取最相关的知识片段，并保证结果附带元信息使答案可溯源[17][18]。

- **根因分析与模式推荐接口**：针对性能瓶颈或错误问题，提供专门的分析查询和推荐功能。当Agent检测到某种问题迹象（例如某帧耗时激增，或者出现特定错误代码），可以调用此接口获取**可能的根因/解决方案**。实现上，该接口会根据输入的问题描述或事件ID，在知识库中执行更深层的检索和逻辑：

- 首先在**事件索引**中查找历史上**相似**的事件发生场景。例如输入是"帧耗时突然增高"，系统可以搜索历史事件中帧耗时突增的记录，筛选条件包括相似的环境（关卡、硬件）和相似的症状（CPU或GPU耗时激增）。借助向量相似度，可以找到哪几次事件与当前情况在文本描述上最相近 [19]。
- 然后获取这些类似事件关联的**总结/后记**（如当时的PerformanceSummary或事后分析），提取其中记录的原因和解决措施。如果知识库里已有"模式"，例如"每当大量Physics刚体生成时帧率骤降，这是物理线程瓶颈，可通过启用异步撕裂缓解"，那么该模式将以Summary形式存在，被检索出来提供给Agent。

- 最终接口返回**候选根因列表**（可能按置信度排序），每个包含简要说明和引用资料来源。Agent 可以据此向用户建议可能的原因和解决方案，甚至自动执行一些验证步骤。
  此接口本质上是对知识库做**关联推理**：把当前问题投射到已存知识中寻找匹配。如果有足够积累，它能将**经验模式**推荐出来。随着数据丰富，系统可不断学习常见问题模式，提高推荐准确率。

- **模型微调与训练接口**：为了让AI Agent 更加熟悉项目的专有数据，可能需要对LLM进行一定程度的微调或训练。知识基础设施可以提供两方面支持：

- **数据导出接口**：能够按需导出某类数据集用于训练。例如导出所有错误日志及对应的总结分析，形成问答对；或导出若干版本的性能数据与优化结果，以fine-tune模型对性能诊断的理解。这种接口需支持过滤和格式转换，把知识库内容整理成训练所需的格式（如JSONL问答对、对话脚本等）。

- **在线学习接口**（可选）：允许Agent在运行中将新经验反馈入库，并适时调整自身参数。例如Agent解决了一个新问题，可以把问题描述与解决方案作为新的Summary存档，同时标记此对话日志训练样本用于日后模型继续微调。
  然而，需要注意微调大模型通常成本高、风险大（可能遗忘原有知识）。在多数情况下，我们会优先采用RAG而非全面微调来赋能Agent [20] [21]。因此，这部分接口更多是为**小型专用模型**或特定模块服务。例如，可以训练一个分类模型先预测提问属于"性能问题"还是"错误排查"，这个模型训练数据就来自知识库标注的历史问题类型。

- **工具与可执行接口**：除了查询数据，本系统也可以提供Agent调用的工具操作接口。例如：

- `get_latest_trace(session_id)`：获取某次运行的trace文件供Agent分析（如果未来Agent具备自行运行分析脚本的能力）。
- `open_blueprint(name)`：在编辑器中打开一个蓝图（如果Agent与编辑器集成，可以通过命令触发界面操作，让开发者快速导航到相关蓝图）。
- `diff_config(versionA, versionB)`：输出两个版本配置差异的报告。
  这些接口超出了知识查询的范畴，但有了数据基础设施，开发类似工具会更便利，可视为Agent能调用的"技能"。在规划阶段，可列出一批可能的工具接口，随着系统成熟逐步实现。

接口规划总体遵循**高内聚、低耦合**原则：知识库提供清晰的问答检索服务和数据获取服务，AI Agent 作为客户端调用这些服务并在其逻辑中编排结果。这样一来，即使未来更换底层LLM或增加新数据源，接口契约保持不变，确保Agent功能平稳演进。同时，通过提供引用和元数据，Agent 的回答可保证可验证性和实时性——这在调试场景下至关重要 [18]（答案必须精准且基于最新数据，否则可能误导调试方向）。

建议的 MVP 路线图

构建如此全面的"记忆系统"是一个复杂工程，建议采取**增量迭代**策略。下面提供一个最小可行产品（MVP）的实施路线，聚焦最高优先级的数据和问题：

1. **聚焦性能Trace的初始采集与分析**：首先从最核心的 **Unreal Insights Trace 数据** 入手，这是性能优化的关键。一开始选取项目中一个典型场景（例如一个复杂关卡）进行性能剖析。手工运行一次带Trace的会话，收集 `.utrace` 和日志文件。开发一个简单解析脚本将该Trace中的**关键事件**提取出来（例如帧时间序列、主要线程的开销统计）。验证解析结果的正确性并存入临时数据库。**目标**：证明我们能从Trace自动获取有价值的信息，例如"平均帧耗时Xms，瓶颈在GPU"之类。

2. **建立事件/工件存储雏形**：将上述Trace解析的数据按设计的事件Schema存入关系数据库表中，并设计出基础的查询手段。例如，可以查询某帧范围内CPU占用最高的函数事件。并行地，存储此次场景的配置信息和静态数据（如蓝图数量、Actor数量）作为工件记录。虽然此时数据量小，但已开始形成**事件-工件**的关联（比如这次性能数据对应哪个关卡蓝图）。**目标**：搭建起数据库schema和对象存储目录，确保可以正确写入和读取。

3. **实现基础检索接口**：构建一个简单的API或脚本接口，能够回答一些硬编码的问题。例如"当前场景的平均FPS是多少？"、"最大的CPU瓶颈在哪个函数？"通过查询数据库得到答案。这个阶段可以不引入LLM，先用固定查询验证数据是否支撑这些问题的回答。如果可以，将结果格式化（带上来源，比如"数据来自Session-20231101"）。**目标**：验证知识库内容的**可用性**——也就是存进去的数据能被检索、计算出实用的信息。

4. **增量添加日志错误数据**：在性能Trace跑通后，扩展采集范围到**错误日志**。选择项目中曾出现的一两个典型错误或崩溃，在受控环境重现，收集日志。将错误日志解析为事件，并编写相应的Summary（如该错误产生原因和解决办法的描述）。把这些也存入库中。此时可以尝试引入**向量检索**组件：将错误日志和性能摘要的文本都生成embedding，存入一个简易向量索引（哪怕用本地FAISS）。**目标**：实现一次基于语义的查询测试，例如输入一个接近错误日志描述的自然语言问题，系统能够找到匹配的日志记录或总结说明。这验证了RAG的关键技术路径。

5. **打造初步AI问答原型**：将上一步的检索结果接入一个基础的LLM（可使用小型开源模型本地部署以降低复杂性）。设计几个演示用例的问题，由系统检索知识库片段，然后让LLM生成回答。例如："为什么地图加载后帧率下降？" -> 检索到PerformanceSummary提到因为光源太多 -> LLM整合为答案。又如"如何解决材质编译崩溃？"-> 检索到某错误Summary -> LLM复述解决步骤。 **目标**：验证端到端地**用知识库辅助回答**的可行性，发现检索或回答中的不足（例如某些查询找不到内容，提示我们需要补充哪类数据）。

6. **迭代和拓展**：在MVP验证基础上，逐步拓展系统能力：

7. **更多数据源**：接入**蓝图结构**数据，尤其是与性能和错误强相关的（如复杂的蓝图Tick可能导致性能问题）。采集蓝图的节点数、循环、事件tick频率等作为工件属性，让Agent可以回答"哪个蓝图最耗费帧时间？"这类问题。
8. **跨版本对比**：当项目有新版本发布，再跑性能分析，利用系统比较两次运行数据。MVP 可以先手动触发一个Comparison Summary记录差异。确保知识库能存储多版本数据并区分检索范围。
9. **模式积累**：每解决一个问题，就把问题和解决方法整理成Summary存入库（哪怕人工撰写）。随着积累，Agent 下次遇到类似问题就能检索到现成答案，实现真正的知识复用。

10. **管线自动化**：将最初手动的采集-解析过程用脚本串联，做到一键运行特定关卡并自动完成数据入库和摘要。这为大规模应用打基础。

在这过程中优先解决**高价值场景**。建议优先选择**性能瓶颈分析**作为切入点，因为: - 性能数据通过Trace易于结构化，且与**引擎优化**直接相关，价值高。 - 该领域模式明显（CPU/GPU/内存瓶颈类型有限），易于总结经验。例如，通过几次Trace，我们也许就能总结出3-5种常见瓶颈模式并存入知识库供Agent套用。

一旦性能"记忆"系统初见成效，再扩充到**错误诊断**领域，最后再考虑更复杂的蓝图逻辑问题等。这样的MVP路线，既快速产出可用成果，又为后续更全面的AI Agent辅助手段打下数据基础。

重点回顾：整个方案充分利用 Unreal Insights 提供的结构化追踪能力，将庞杂的运行数据提炼为标准化的事件和知识。通过跨平台、跨版本的一致数据架构，实现对项目演进过程的全面"记忆"。最终，这些记忆将赋能AI Agent，为开发者提供检索增强的智能支持，从而更快地诊断性能瓶颈和错误根因，优化迭代效率 [18] 。

参考文献：

- Unreal Engine 文档: 虚幻引擎中的 *Unreal Insights* [1] [2]
- DebuggAI 博文: *RAG for Debugging AI: Turning Logs, Runbooks, and Incidents into Context-Aware Fixes* [5] [12] [7]
- 腾讯新闻: 告别传统 *RAG*，用智能 *Agent* 方法构建 *AI* 知识库 [18]

---

[1] [2] 虚幻引擎中的Unreal Insights | 虚幻引擎 5.7 文档 | Epic Developer Community
https://dev.epicgames.com/documentation/zh-cn/unreal-engine/unreal-insights-in-unreal-engine

[3] [4] 虚幻开发游戏分析性能瓶颈_unreal insights memory insights 查看character数量-CSDN博客
https://blog.csdn.net/X_Bai01/article/details/146298325

[5] [6] [7] [10] [11] [12] [13] [14] [15] [16] [17] [19] RAG for Debugging AI: Turning Logs, Runbooks, and Incidents into Context-Aware Fixes | DebuggAI Resources
https://debugg.ai/resources/rag-for-debugging-ai-logs-runbooks-incidents-context-aware-fixes

[8] [9] [18] [21] 告别传统 RAG，用智能 Agent 方法构建 AI 知识库_腾讯新闻
https://news.qq.com/rain/a/20251002A06NDL00

[20] 什么是AI 中的检索增强生成(RAG)？ - Cloudflare
https://www.cloudflare.com/zh-cn/learning/ai/retrieval-augmented-generation-rag/