## Module 3: LLD for GenAI

### Unit 1

### Data Ingestion & Preprocessing (Loaders, Chunking)

### 1. Context: Why Data Ingestion & Preprocessing Matter in GenAI LLD

In any **production-grade Generative AI system**, the quality, consistency, and structure of ingested data directly determine the system’s retrieval accuracy, reasoning performance, and overall user trust. While high-level design (HLD) defines where ingestion sits in the architecture, **low-level design (LLD)** addresses how each data ingestion and preprocessing step is implemented — from file reading and format normalization to semantic chunking and metadata enrichment.

In Retrieval-Augmented Generation (RAG), Agentic AI, or Multi-Modal systems, **improper ingestion logic can cascade into model inefficiencies**:

* Poor chunking → retrieval mismatch → hallucinations.
* Inconsistent metadata → inaccurate filtering and slower searches.
* Overly large chunks → token wastage, higher costs, and degraded latency.

An LLD approach ensures ingestion pipelines are **deterministic, modular, and optimized** for:

* **Caching efficiency** — avoiding reprocessing the same data unnecessarily.
* **Batching efficiency** — embedding or indexing multiple chunks in grouped operations.
* **Model-compatibility** — ensuring chunk size and structure match the downstream model’s optimal context window.

### 2. Core Components of Data Ingestion & Preprocessing

#### 2.1 Data Loaders

**Role in LLD:**  
Loaders are specialized modules or services that read raw input from various sources and transform it into an **internal canonical format** ready for preprocessing. In GenAI, loaders must handle **heterogeneous file types** and **streaming or real-time feeds** without sacrificing performance or reliability.

**Loader Types:**

* **Static File Loaders:** Handle documents like PDFs, Word files, CSVs, JSON, XML.
* **Web/HTML Loaders:** Scrape or pull content from websites, blogs, APIs.
* **Database Loaders:** Pull structured/unstructured data from SQL/NoSQL sources.
* **Streaming Loaders:** Consume continuous feeds (e.g., Kafka topics, S3 events).

**Design Practices:**

* **Unified Loader Interface:** Implement an abstract BaseLoader class with a common output schema (e.g., content, metadata, source\_id).
* **Incremental Loading:** Support delta ingestion to process only new or modified files.
* **Compliance Hooks:** Integrate PII detection at the loader level to redact sensitive data early.
* **Caching Intersection:**
  + Maintain a **document hash cache** — if the hash hasn’t changed, skip reprocessing.
  + Store pre-cleaned outputs in cache to speed up re-indexing cycles.
* **Batching Intersection:**
  + Group documents for bulk processing in downstream embedding/indexing steps.

**Pitfalls:**

* Tight coupling between loader code and file formats → harder to extend.
* Failing to normalize encodings → downstream chunkers fail on non-UTF-8 text.
* Not handling large files with streaming techniques → memory bloat.

#### **2.2 Text Cleaning & Normalization**

**Role in LLD:**  
Before chunking, text must be **cleaned** (remove irrelevant characters, headers, footers, boilerplate) and **normalized** (consistent spacing, punctuation, casing). This step ensures retrieval relevance and prevents token budget waste.

**Key Cleaning Actions:**

* Remove boilerplate (page numbers, table of contents in PDFs).
* Strip HTML tags or inline scripts for web content.
* Normalize Unicode, fix broken encodings.
* Apply lowercasing selectively (avoid in contexts where casing matters).

**Caching Intersection:**

* Store cleaned text keyed by document hash — avoids re-running expensive regex or NLP cleaning.

**Pitfalls:**

* Over-cleaning → loss of valuable context (e.g., removing section headings that aid retrieval).
* No language detection → applying wrong normalization rules to multi-lingual datasets.

#### **2.3 Chunking Strategies**

**Role in LLD:**  
Chunking is the process of splitting cleaned text into **smaller, semantically coherent units** that fit within the model’s input token limit. This is one of the most **impactful design decisions** in GenAI ingestion pipelines — it directly affects retrieval precision, LLM reasoning quality, caching granularity, and embedding costs.

**Common Chunking Methods:**

* **Fixed Token/Character Length:** Simple, predictable, but may split mid-sentence.
* **Semantic/Paragraph-Aware:** Uses NLP to split along natural boundaries (sentences, paragraphs).
* **Hybrid Chunking:** Combines length constraints with semantic boundaries.
* **Adaptive Chunking:** Dynamically adjusts chunk size based on content density and model context window.

**Design Practices:**

* Always maintain **chunk overlap** (e.g., 50 tokens) to preserve context continuity.
* For legal, technical, or policy documents, prefer **semantic or section-based** chunking to preserve meaning.
* Keep chunk size **aligned with embedding model’s optimal range** (often 256–512 tokens for balance of precision and cost).

**Caching Intersection:**

* Cache embeddings per chunk hash — if text doesn’t change, reuse the embedding.

**Batching Intersection:**

* Batch embeddings for multiple chunks into a single API call to reduce latency and API costs.

**Pitfalls:**

* Too small chunks → high recall but low precision (model lacks sufficient context).
* Too large chunks → retrieval returns fewer, less targeted results, wasting tokens.
* Ignoring document structure → retrieval surfaces irrelevant fragments.

#### **2.4 Metadata Enrichment**

**Role in LLD:**  
Metadata provides **contextual labels** for chunks, enabling **filtered retrieval** and better ranking.

**Common Metadata Fields:**

* source\_id – document identifier.
* section – document section name or heading.
* timestamp – publication or ingestion date.
* jurisdiction – for legal/compliance retrieval.

**Design Practices:**

* Attach metadata **before embedding** so it’s stored alongside the vector representation.
* Standardize field names and formats for cross-system compatibility.

**Caching Intersection:**

* Cache metadata lookups in memory or Redis to accelerate repeated queries.

**Pitfalls:**

* Inconsistent metadata keys → retrieval filters fail.
* Excessively large metadata blobs → increased storage and network overhead.

### 3. Design Considerations in LLD

* **Idempotency:** Ensure ingestion is repeatable without producing duplicates.
* **Parallelization:** Use worker pools for large ingestion jobs but avoid overloading downstream embedding services.
* **Observability:** Log ingestion metrics — processing time, chunk counts, skipped documents due to cache hits.
* **Security:** Encrypt data in transit and at rest during ingestion, especially in compliance-heavy domains.

### 4. Example Scenario – Policy Document Ingestion for RAG

**Use Case:** Corporate PolicyRAG assistant that ingests updated policy manuals.

**Implementation Highlights:**

* **Loaders:** Detect changed files in S3 via event triggers, load PDF text using streaming parser, store cleaned text in cache.
* **Cleaning:** Remove footer disclaimers, normalize bullet lists, preserve section titles.
* **Chunking:** Apply semantic chunking with 50-token overlaps, aligned to max 400 tokens per chunk for embedding.
* **Metadata:** Attach policy\_version, department, jurisdiction for retrieval filtering.
* **Caching:** Skip embedding for unchanged chunks using SHA256 hash matching.
* **Batching:** Send 64 chunks at a time to the embedding API for optimal throughput.

### 5. Key Takeaways

* **Loaders** must abstract data source complexity and output a consistent schema.
* **Cleaning & Normalization** prevent token waste and improve retrieval relevance.
* **Chunking** is a **high-leverage step** — design it with model limits, retrieval quality, and cost trade-offs in mind.
* **Metadata** enables precise retrieval, but requires consistency in keys and formats.
* **Caching & Batching** should be integrated early in LLD to avoid wasted computation and cost.
* Observability and compliance hooks are not optional — they prevent silent data quality degradation and legal exposure.

### Unit 2: Embedding & Vector Store Layer

### (FAISS, Pinecone, Weaviate, Qdrant, Milvus, Chroma, Redis)

### 1. Context: Why the Embedding & Vector Store Layer is Critical in GenAI LLD

In **retrieval-augmented architectures**, the embedding and vector store layer is the **memory backbone**. It converts text (or other modalities) into dense vector representations and stores them for **fast, semantically relevant retrieval**.

In Low-Level Design (LLD), this layer is not just about choosing a vector database. It’s about **fine-grained implementation details** that directly affect:

* **Retrieval precision & recall** (driven by embedding model choice and chunk strategy).
* **Cost efficiency** (optimized batching & caching of embeddings).
* **Latency performance** (index type, storage backend, network topology).
* **Compliance** (geo-location storage restrictions, secure embedding handling).

A poorly designed embedding & vector store layer will result in:

* Irrelevant retrievals → model hallucinations.
* High costs from redundant embedding generation.
* Slow query performance, frustrating users and breaking SLAs.

### 2. Core Components of the Embedding & Vector Store Layer

#### **2.1 Embedding Model Selection**

**Role in LLD:**  
The embedding model transforms text into numerical vectors capturing semantic meaning. The choice of model determines **retrieval quality, inference cost, and compatibility** with downstream systems.

**Key Considerations:**

* **Embedding Dimensionality:** Higher dimensions capture richer semantics but increase storage and compute costs.
* **Domain Specialization:** Use domain-tuned models (e.g., biomedical embeddings for healthcare).
* **Vendor & Ecosystem:**
  + OpenAI (text-embedding-3-small / text-embedding-3-large) for high precision.
  + Hugging Face models for on-prem / cost control.
  + Cohere embeddings for multilingual use.
* **Update Frequency:**
  + Models evolve; plan for re-embedding if upgrading.
  + Store original text alongside embeddings for future reprocessing.

**Caching Intersection:**

* Always hash chunks (SHA256) and cache embeddings keyed by hash to avoid recomputing unchanged data.

**Batching Intersection:**

* Send multiple chunks in a single API call to reduce request overhead and improve throughput.

**Pitfalls:**

* Using a general-purpose model for niche domains → poor retrieval relevance.
* Ignoring embedding size when selecting vector DB → degraded performance from oversized vectors.

#### **2.2 Vector Store Selection & Indexing**

**Role in LLD:**  
The vector store organizes embeddings into a searchable index that can return the most semantically similar vectors to a query embedding.

**Popular Options:**

* **FAISS** – Local, high-performance, great for batch/offline retrieval; lacks built-in distributed scaling.
* **Pinecone** – Fully managed, scalable, low-latency, rich metadata filtering.
* **Weaviate** – Schema-based, hybrid search (vector + keyword), good for multi-modal.
* **Qdrant** – Open-source, powerful filtering, REST + gRPC APIs.
* **Milvus** – Distributed, suited for massive-scale vector workloads.
* **Chroma** – Simple local development store, rapid prototyping.
* **Redis (Vector Search)** – Adds ANN search to Redis for integrated caching + retrieval.

**Design Practices:**

* Match **index type** (HNSW, IVF, Flat) to use case:
  + **HNSW:** High recall, fast queries for real-time.
  + **IVF:** Scales to billions of vectors, good for large datasets.
* Optimize **metadata schema** for filtering at retrieval time.
* Consider **sharding strategy** for distributed stores.

**Caching Intersection:**

* Cache query results for frequent searches, but be cautious with time-sensitive domains.

**Batching Intersection:**

* Bulk insert embeddings in batched writes to avoid per-vector write overhead.

**Pitfalls:**

* Not enabling persistence in in-memory vector DBs → data loss on restart.
* Overloading a single index with multiple unrelated domains → retrieval noise.

#### **2.3 Metadata Storage & Filtering**

**Role in LLD:**  
Metadata enables **precise retrieval filtering** and hybrid search strategies.

**Best Practices:**

* Store metadata alongside vectors at insertion time.
* Normalize field names and types (string, int, datetime) for query consistency.
* Use metadata for compliance filters (jurisdiction, document type).

**Caching Intersection:**

* Cache frequent metadata filter queries to accelerate retrieval.

**Pitfalls:**

* Overly complex metadata schemas slowing down queries.
* Missing indexes on metadata fields → increased latency.

#### **2.4 Query Workflow & Similarity Search**

**Workflow Steps:**

1. Convert user query into embedding using the same model as document embeddings.
2. Perform **Approximate Nearest Neighbours (ANN)** search for speed, or **exact search** for small datasets.
3. Apply metadata filters to refine results.
4. Return top-k results with associated text for LLM prompt assembly.

**Performance Enhancements:**

* **Re-ranking:** Apply a lightweight cross-encoder to refine top results.
* **Query Expansion:** Add synonyms or related terms before embedding.

**Caching Intersection:**

* Cache embeddings for repeated queries to skip re-embedding.

**Batching Intersection:**

* If processing multiple related queries, batch embeddings before search.

**Pitfalls:**

* Using a different embedding model for queries vs documents → retrieval quality collapse.
* Not tuning k (number of results) — too high wastes tokens, too low risks missing context.

### 3. Design Considerations in LLD

* **Consistency:** Same embedding model for indexing and querying.
* **Storage & Cost:** Factor in vector dimensionality × number of chunks × replication.
* **Latency SLAs:** Choose index type and hosting location to meet performance requirements.
* **Geo-Compliance:** Keep vectors in region-specific databases for jurisdiction compliance.
* **Observability:** Track query latency, retrieval hit rate, and cache utilization.

### 4. Example Scenario – PolicyRAG Vector Store

**Use Case:** Corporate policy assistant using Qdrant for semantic retrieval.

**Implementation Highlights:**

* **Embedding Model:** text-embedding-3-large for rich semantic capture of legal documents.
* **Caching:** Redis layer keyed by chunk hash to skip redundant embeddings.
* **Batching:** Group 64 chunks per embedding API call.
* **Vector Store:** Qdrant with HNSW index, metadata fields policy\_version, jurisdiction.
* **Query Flow:** Query embedded → ANN search with jurisdiction filter → top 5 chunks → sent to LLM.
* **Compliance:** EU policies stored in EU-hosted Qdrant cluster.

### 5. Key Takeaways

* **Embedding model choice** is a major determinant of retrieval quality — match it to domain and context window needs.
* **Vector store selection** should balance scale, latency, filtering capability, and operational cost.
* **Caching & batching** are essential to prevent unnecessary API calls and reduce latency.
* **Metadata filtering** improves retrieval precision but requires schema discipline.
* A robust query workflow ensures consistency, performance, and compliance in production GenAI systems.

### Unit 3

### Prompt Orchestration Layer

### (Templates, Dynamic Assembly)

### 1. Context: Why the Prompt Orchestration Layer Matters in GenAI LLD

In any **Generative AI system**, the prompt orchestration layer acts as the bridge between raw application intent and optimal Large Language Model (LLM) interaction. It’s responsible for **structuring, enriching, and dynamically assembling** inputs to maximize LLM accuracy, consistency, and compliance — all while keeping latency, cost, and maintainability in check.

At the LLD stage, this layer is not just “how prompts are written” — it’s **how prompt generation logic is engineered, modularized, and integrated** with other components like retrieval, validation, and caching.  
A well-designed prompt orchestration layer ensures:

* Reusability of templates across flows.
* Consistency in schema and formatting for model consumption.
* Dynamic injection of relevant context without bloating token usage.
* Controlled variability for experimentation and A/B testing.

Poorly implemented orchestration often results in:

* Unstructured prompts → LLM misinterpretation.
* Token overflows → request failures or truncation.
* Inability to update prompts quickly across large systems.

### 2. Core Components of the Prompt Orchestration Layer

#### **2.1 Prompt Templates**

**Role in LLD:**  
Prompt templates define **structured blueprints** for how instructions, context, and variables are combined before being sent to the model.

**Key Practices:**

* **Separation of Concerns:** Keep template definitions separate from business logic.
* **Parameterization:** Use placeholders ({user\_query}, {context}, {format\_instructions}) for flexible substitution.
* **Versioning:** Maintain template versions to compare performance across iterations.
* **Schema Integration:** Embed JSON schema or structured output guidelines directly into the prompt for consistent downstream parsing.
* **Domain Optimization:** Adapt templates to target domain — compliance, customer service, medical, etc.

**Caching Intersection:**

* Cache compiled templates for static sections to avoid repeated rendering cost.

**Batching Intersection:**

* When processing similar queries, batch template rendering with shared context injection to reduce retrieval overhead.

**Pitfalls:**

* Hardcoding template strings in code → difficult to update or localize.
* Overly verbose templates → higher token cost and slower inference.

#### **2.2 Dynamic Context Assembly**

**Role in LLD:**  
Dynamic assembly is the process of **injecting runtime variables and retrieved context** into templates while preserving structure and compliance.

**Key Steps:**

1. **Retrieve Context:** Gather top-k relevant chunks from vector store or API results.
2. **Transform Context:** Apply summarization, filtering, or ranking.
3. **Insert into Template:** Merge with placeholders while respecting token constraints.
4. **Pre-Validation:** Ensure assembled prompt doesn’t exceed model token limit.

**Advanced Strategies:**

* **Context Prioritization:** Rank retrieved chunks by relevance and insert the most important first.
* **Adaptive Context Windows:** Dynamically adjust number of chunks based on user query complexity.
* **Instruction-Context Separation:** Keep instructions distinct from context to avoid prompt injection attacks.

**Caching Intersection:**

* Cache context-assembled prompts for repeated or follow-up queries.

**Batching Intersection:**

* When sending prompts for multiple related user queries, reuse common retrieved context to reduce retrieval calls.

**Pitfalls:**

* Inconsistent variable substitution → broken prompt logic.
* Context overflow leading to model truncation mid-instruction.

#### **2.3 Multi-Stage Prompting**

**Role in LLD:**  
Instead of sending one large prompt, break the interaction into multiple stages for **accuracy and efficiency**.

**Examples:**

* **Stage 1:** Extract entities from query.
* **Stage 2:** Retrieve documents.
* **Stage 3:** Generate final structured answer.

**Benefits:**

* Reduces cognitive load on the model.
* Improves controllability and reduces hallucination risk.

**Caching Intersection:**

* Cache outputs of earlier stages to skip re-extraction or re-retrieval in follow-ups.

**Pitfalls:**

* Latency stacking if stages are not parallelized or cached.

#### **2.4 Prompt Variants & A/B Testing**

**Role in LLD:**  
Prompt variants allow teams to **experiment** with different phrasings, structures, or ordering of context to see what yields best results.

**Best Practices:**

* Run A/B tests on a small percentage of traffic.
* Measure precision, latency, and user satisfaction.
* Log variant ID with responses for traceability.

**Pitfalls:**

* Testing too many variants without statistical rigor → misleading conclusions.

#### **2.5 Prompt Security & Compliance Guards**

**Role in LLD:**  
Prevent malicious or unintended instructions from propagating to the LLM.

**Key Techniques:**

* **Prompt Injection Detection:** Filter or sanitize context before insertion.
* **Escaping User Input:** Prevent execution of system commands in prompt logic.
* **Compliance Anchoring:** Embed must-follow policy rules at the top of the prompt.

### 3. Design Considerations in LLD

* **Token Efficiency:** Keep prompts as short as possible while maintaining clarity.
* **Traceability:** Log prompts and versions for debugging and compliance audits.
* **Extensibility:** Support adding/removing context sources without breaking templates.
* **Observability:** Monitor prompt length, cache hit rates, and variant performance.

### 4. Example Scenario – PolicyRAG Prompt Orchestration

**Use Case:** A compliance assistant generating structured, citation-backed policy answers.

**Implementation Highlights:**

* **Templates:** Version-controlled YAML templates with placeholders for {question}, {context}, and {required\_fields}.
* **Dynamic Assembly:** Retrieves top 5 chunks from Qdrant, ranks by citation relevance, inserts into context block.
* **Multi-Stage Flow:** Stage 1 – classify query type; Stage 2 – retrieve; Stage 3 – answer with citations.
* **Caching:** Stores rendered prompt for identical queries within 24-hour window.
* **Batching:** Uses shared context injection for multiple related queries from same department.
* **Compliance Guards:** Prepends “You must follow GDPR compliance rules” to all responses involving EU data.

### 5. Key Takeaways

* Prompt orchestration in LLD is about **systematic, reusable, and secure prompt construction.**
* Templates should be parameterized, versioned, and domain-aware.
* Dynamic context assembly must balance relevance with token budget constraints.
* Multi-stage prompting improves control and reduces hallucinations.
* Caching and batching can significantly lower latency and cost in high-traffic systems.
* Security guards against prompt injection and ensures compliance.

### ****Unit 4A****

### ****Agent Pattern – ReAct (Reason + Act)****

### ****1. Context: Why ReAct Matters in GenAI LLD****

Generative AI systems differ fundamentally from traditional deterministic programs. In conventional software, the execution path is fixed: given an input, the system follows a predetermined set of instructions to produce an output. In contrast, Generative AI agents often operate in **dynamic, decision-driven environments** where the best course of action isn’t known in advance and may need to be discovered step-by-step.

The **ReAct pattern**—short for Reason + Act—is a paradigm designed to combine **logical reasoning** and **tool usage** into a single iterative loop. The agent alternates between thinking about what to do and actually doing it, incorporating feedback from each action into subsequent reasoning steps.

In Low-Level Design (LLD), the ReAct pattern takes on heightened significance because:

* It introduces **statefulness** into LLM interactions.
* It requires careful **prompt structuring** to enforce reasoning-action separation.
* It demands **tool integration with safety controls** to avoid undesired behaviors.
* It benefits from **performance optimizations** such as model selection, caching, and batching.

In production-grade GenAI architectures, ReAct enables agents to:

* Tackle **multi-step reasoning problems** that cannot be solved in a single prompt.
* Integrate seamlessly with APIs, databases, and other external systems.
* Adapt mid-process if new information changes the problem landscape.
* Provide **explainable outputs** by exposing their chain of thought and actions.

Without a well-engineered ReAct loop, agents risk becoming inefficient, unsafe, and costly to operate—burning through tokens, repeating actions unnecessarily, or triggering tools incorrectly.

### ****2. The Core Loop of the ReAct Pattern****

A ReAct agent operates in a continuous loop that can be broken down into distinct phases:

1. **User Input**  
   The agent receives a query, goal, or instruction. This may be accompanied by contextual metadata, constraints, or a desired output format.
2. **Reasoning Step**  
   The LLM generates a rationale—its internal thought process on what the next logical move should be. This is explicitly separated from the final answer to avoid premature conclusions.
3. **Action Step**  
   The model outputs a structured instruction to execute a tool, query a database, or perform another external operation.
4. **Tool Execution**  
   The system receives the action, validates it against a schema, and executes the requested operation.
5. **Observation Step**  
   The result from the tool is recorded in the agent’s working memory (“scratchpad”).
6. **Loop Decision**  
   The LLM determines whether another Reason–Act cycle is needed or if it can produce the final answer.
7. **Termination**  
   The loop ends either when the final answer is ready, a maximum step limit is reached, or a fail condition is triggered.

In LLD terms, each of these phases maps to a **control point** where architectural and engineering decisions can be made to improve reliability, cost efficiency, and safety.

### ****3. Detailed Components of a ReAct Implementation****

#### **3.1 Controller Logic**

The controller is the orchestration layer that governs the ReAct loop. It:

* Enforces **maximum step limits** to avoid infinite loops.
* Monitors for **no-progress scenarios**, where actions repeat without advancing the solution.
* Implements **escalation rules** for model switching when complexity increases.
* Tracks performance metrics such as step latency and token usage.

**Design Considerations:**

* Use a **finite-state machine** model with explicit states: REASON, ACT, OBSERVE, TERMINATE.
* Implement **guard conditions** to exit loops early if the agent reaches a dead end.
* Allow **configurable limits** so different tasks can have different maximum iteration counts.

#### **3.2 Prompt Template**

A well-engineered prompt is the foundation of the ReAct pattern. The system message must clearly define:

* The **agent’s role and scope**.
* The **list of available tools** and their purposes.
* The **format for reasoning and actions**, ideally in a structured schema (e.g., JSON).
* Guidelines for **concise reasoning** to manage token usage.

**Example Prompt Structure:**

You are a compliance research assistant.

For each step:

1. Output your reasoning in "rationale".

2. Output your "action" as a TOOL\_NAME or "final\_answer".

3. If action is a tool, include "arguments" as JSON.

#### **3.3 Tool Adapter Layer**

Tools are the bridge between the LLM and the outside world. The adapter layer:

* **Validates inputs** using strict schema enforcement (Pydantic or similar).
* **Normalizes outputs** into a consistent format.
* Applies **security controls**, such as authentication and rate limiting.
* Handles **errors gracefully**, returning structured error messages to the LLM.

**Implementation Notes:**

* Maintain a **tool registry** mapping tool names to callable functions and schemas.
* Include **logging hooks** for every tool invocation to support audit trails.
* Add **timeout enforcement** to prevent long-running calls from blocking the loop.

#### **3.4 Observation Memory**

The scratchpad stores the history of reasoning steps, actions, and tool outputs.

* Without memory, the LLM would have no continuity between steps.
* The memory must balance **context completeness** with **token efficiency**.

**Strategies:**

* **Summarize older steps** to free up token space.
* **Pin critical facts** so they are never lost during summarization.
* Use a **structured format** for observations so they can be parsed and referenced reliably.

### ****4. Performance Engineering: Model Selection, Caching, and Batching****

#### **4.1 Model Selection**

Not all reasoning steps require the same model:

* **Small, fast models** handle routine reasoning cheaply.
* **Larger models** are used selectively for complex decision-making.
* Implement **dynamic routing** based on step complexity and confidence scores.

#### **4.2 Caching**

Caching is critical to avoid redundant tool calls:

* Use deterministic cache keys derived from (tool\_name, arguments).
* Implement a **time-to-live (TTL)** to keep results fresh.
* Consider **layered caches** (in-memory for recent calls, Redis for persistent storage).

#### **4.3 Batching**

Although ReAct is sequential by nature, batching can reduce latency when multiple independent lookups are needed.

* Detect **parallelizable actions** during the reasoning step.
* Group them into a single batch call when possible.
* This is particularly effective for **vector database queries** and API fetches.

### ****5. Safety and Guardrails****

Safety measures are essential for production-grade ReAct agents:

* **Input validation** for every tool call.
* **Prompt injection defense** to filter malicious user input.
* **Schema enforcement** to reject malformed actions.
* **Compliance filtering** to enforce domain-specific rules.

### ****6. Observability and Monitoring****

Robust observability ensures that ReAct loops remain maintainable and debuggable:

* Log **every step** with reasoning, action, and observation data.
* Capture **latency metrics** for each stage.
* Monitor **loop termination reasons** for signs of inefficiency.
* Store traces for offline evaluation and model improvement.

### ****7. Example: Policy Compliance Research Agent****

**Scenario:**  
A compliance officer asks for the most recent EU AI Act guidelines on “data retention.”

**ReAct Flow:**

1. Reason: Decide to search legal database.
2. Act: Call search\_documents.
3. Observe: 3 relevant clauses returned.
4. Reason: Choose the most relevant clause for extraction.
5. Act: Call extract\_clause\_details.
6. Observe: Full clause text returned.
7. Reason: Decide information is complete.
8. Final Answer: Provide formatted clause with citation.

This flow completes in **two tool calls and three reasoning steps**, with search results cached for reuse.

### ****8. Common Pitfalls and Anti-Patterns****

* **Infinite loops** due to missing step limits.
* **Token bloat** from verbose reasoning.
* **Repeated tool calls** without caching.
* **Loose schemas** causing unpredictable outputs.
* **Unbounded scratchpad growth** leading to context overflow.

### ****9. Key Takeaways****

* The ReAct pattern enables **multi-step reasoning with tools**, but requires strong LLD discipline.
* Every stage of the loop is a **control point** for optimization and safety.
* **Model selection, caching, and batching** drive the biggest performance gains.
* Guardrails must be applied **at every step**, not just at the end.
* A well-engineered ReAct loop is explainable, adaptable, and production-ready.

### ****Unit 4B****

### ****Agent Pattern – Plan-and-Execute****

### ****1. Context: Why Plan-and-Execute Matters in GenAI LLD****

The **Plan-and-Execute** pattern is designed for scenarios where an agent must break a complex task into multiple sub-tasks, determine their execution order, and carry them out step-by-step. Unlike the **ReAct** pattern, which mixes reasoning and action in a single loop, Plan-and-Execute **front-loads the reasoning** by creating a structured plan before executing any steps.

This separation between **planning** and **execution** offers several benefits in Low-Level Design:

* The agent can **optimize the overall sequence** before starting work, reducing wasted effort.
* Plans can be **reviewed, adjusted, or approved** before execution, which is especially valuable in compliance-heavy domains.
* Execution can be **delegated to different models, services, or micro-agents** that specialize in each sub-task.

Plan-and-Execute is a strong fit for GenAI use cases such as:

* Research agents compiling structured reports.
* Multi-document summarization pipelines.
* Complex multi-modal workflows (e.g., text + image generation).
* Scenarios requiring human-in-the-loop approval before final execution.

### ****2. Core Phases of the Plan-and-Execute Pattern****

#### **2.1 Planning Phase**

The first phase involves generating a high-level plan:

* **Input**: The user’s query or goal, possibly with constraints, priorities, or deadlines.
* **Process**:
  + The LLM acts as a planner, breaking down the goal into discrete, actionable steps.
  + Each step includes **description, expected input/output, dependencies, and tool requirements**.
* **Output**: A structured execution plan, often in JSON or YAML format for machine readability.

**Engineering Considerations**:

* Use **structured prompts** to enforce consistent step formats.
* Allow **configurable planning depth** — simple tasks may need 2–3 steps, complex ones may require 10+.
* Incorporate **domain-specific constraints** during planning to ensure feasibility.

#### **2.2 Execution Phase**

Once the plan is ready, the agent executes each step in order:

* **Execution Controller**: Iterates through the plan, invoking tools, services, or sub-agents.
* **Progress Tracking**: Logs outputs and marks each step as completed.
* **Error Handling**: Decides whether to retry, skip, or escalate failures.

**Engineering Considerations**:

* Maintain **state persistence** between steps to allow pause/resume.
* Use **dynamic model selection** — smaller models for simple steps, larger ones for complex reasoning.
* Implement **caching** to avoid repeating identical tool calls across steps.

### ****3. LLD Components of Plan-and-Execute****

#### **3.1 Plan Generator Module**

* **Purpose**: Produces a structured execution plan from the user’s query.
* **Prompting Strategy**:
  + Role definition: e.g., “You are an AI planner…”
  + Clear output schema for each plan step.
* **Validation**: Apply JSON schema checks to ensure the plan is syntactically correct.
* **Enhancements**:
  + Add **scoring logic** to rank steps by priority.
  + Include **estimated costs or execution times**.

#### **3.2 Execution Engine**

* **Purpose**: Orchestrates the actual execution of the plan.
* **Design**:
  + Reads the plan sequentially or in parallel (if steps are independent).
  + Integrates with **Tool Adapter Layer** for tool calls.
  + Supports **conditional branching** if the plan allows decision-making mid-execution.
* **Logging**: Store both intermediate outputs and final results for auditing.

#### **3.3 Observation Memory**

* Stores step outputs for use in later steps.
* Allows the agent to **reference earlier results** without re-running them.
* Should be **token-efficient** — summarize large outputs when possible.

#### **3.4 Tool Adapter Layer**

* Similar to ReAct’s adapter layer, but optimized for **predictable, pre-planned calls**.
* Validation can be stricter because the plan specifies the required tool inputs in advance.

### ****4. Performance Engineering in Plan-and-Execute****

#### **4.1 Model Selection**

* Use **fast, low-cost models** for simple retrieval or formatting tasks.
* Reserve **large, high-context models** for heavy reasoning steps like initial planning or synthesis.
* Implement **routing logic** to map plan steps to the optimal model.

#### **4.2 Caching**

* Cache **both tool outputs and step results**.
* Reuse intermediate results if the same step is repeated in similar contexts.
* Maintain a **cache invalidation policy** for time-sensitive data.

#### **4.3 Batching**

* Detect when multiple plan steps involve similar queries (e.g., fetching multiple documents) and execute them in a single batched call.
* This reduces API round-trips and latency.

### ****5. Safety and Guardrails****

* **Plan Validation**: Ensure the plan does not include unsafe or disallowed actions.
* **Execution Constraints**: Limit tool calls to approved domains or data sources.
* **Output Filtering**: Apply compliance filters to final outputs.
* **Step-level Guardrails**: For sensitive steps, enforce manual approval before execution.

### ****6. Example: Research Report Generation Agent****

**User Input**:  
"Generate a comparative analysis of privacy laws in the EU, US, and India, focusing on AI data retention."

**Plan**:

1. Search legal databases for relevant laws.
2. Extract relevant sections for each jurisdiction.
3. Summarize each jurisdiction’s laws in plain English.
4. Compare and contrast laws, highlighting overlaps and conflicts.
5. Generate a formatted report.

**Execution**:

* Steps 1–3 executed using retrieval tools and summarization models.
* Step 4 executed by a high-accuracy model.
* Step 5 handled by a formatting tool integrated with a document generator.

### ****7. Common Pitfalls****

* **Over-planning**: Creating unnecessarily detailed plans wastes time and tokens.
* **Rigid execution**: Failing to adapt if circumstances change mid-execution.
* **Token waste**: Keeping full plan context in every step instead of summarizing.
* **Unclear dependencies**: Steps fail because prerequisite data was not properly captured.

### ****8. Key Takeaways****

* Plan-and-Execute is best for **structured, multi-step tasks** where the steps can be clearly defined in advance.
* Separating planning from execution enables **review, optimization, and delegation**.
* Strong **model selection, caching, and batching** strategies are critical for efficiency.
* Guardrails must be applied **both at the planning stage and the execution stage**.
* Ideal for compliance-heavy, research-intensive, and multi-agent GenAI workflows.

### ****Unit 4C****

### ****Agent Pattern – Negotiation****

### ****1. Context: Why Negotiation Matters in GenAI LLD****

The **Negotiation pattern** in GenAI refers to the orchestration of **multiple agents** (human or AI) that engage in structured back-and-forth exchanges to reach a mutually acceptable decision, solution, or output.  
While patterns like **ReAct** and **Plan-and-Execute** focus on problem-solving within a single agent’s reasoning space, Negotiation is inherently **multi-agent**—each participant has its own perspective, data, and objectives.

In **Low-Level Design (LLD)**, negotiation becomes critical when:

* Multiple specialized AI agents must **align outputs** (e.g., legal advisor agent vs. cost optimization agent).
* Trade-offs must be resolved in **ambiguous or multi-objective contexts**.
* There’s a need to **simulate human-like debate or bargaining** to converge on optimal results.
* A system must reconcile **conflicting constraints**, such as compliance requirements vs. speed or cost.

Negotiation in production-grade GenAI systems often appears in:

* Multi-domain RAG systems where retrieved facts may conflict.
* AI-assisted procurement tools that evaluate vendor proposals.
* Multi-LLM setups for **self-evaluation and refinement** of generated content.
* Dynamic decision-making systems in **autonomous operations** (finance, logistics, healthcare).

### ****2. Core Phases of the Negotiation Pattern****

#### **2.1 Participant Definition**

* Each agent must have a **clear role, goal, and domain expertise**.
* Example: In a travel planning AI, the **Budget Agent** optimizes for cost, while the **Experience Agent** optimizes for quality.
* LLD must include **capabilities matrix** per agent—what data they access, what tools they can use, and how they can interact.

#### **2.2 Proposal Phase**

* Agents submit **initial proposals** based on their objectives and available information.
* These proposals follow a **standard schema** for consistency (e.g., {proposal\_text, justification, score}).

#### **2.3 Critique & Counter-Proposal Phase**

* Each agent evaluates others’ proposals and returns:
  + Strengths.
  + Weaknesses.
  + Suggested modifications.
* This can be **iterative**, with multiple rounds until convergence.

#### **2.4 Convergence & Resolution**

* The negotiation loop ends when:
  + A predefined **agreement threshold** is reached.
  + A **timeout or iteration cap** occurs.
  + A human-in-the-loop finalizes the decision.

#### **2.5 Execution**

* The agreed-upon plan, recommendation, or action is **passed downstream** to execution layers or returned to the user.

### ****3. LLD Components for Negotiation****

#### **3.1 Negotiation Orchestrator**

* Governs the negotiation flow.
* Maintains a **state machine** with phases like INITIATE, PROPOSE, CRITIQUE, RESOLVE.
* Applies **coordination rules**, such as turn-taking or simultaneous submissions.

#### **3.2 Communication Protocol**

* Defines the **data structure** for messages between agents.
* Ensures **context continuity**—participants must see relevant prior exchanges without overwhelming the context window.
* Example: Summarize earlier rounds but **pin key facts**.

#### **3.3 Agent Specialization**

* Each agent uses a **domain-specific prompt template**.
* LLD ensures **tool binding** so agents only use permitted tools/data.
* Supports **model selection per agent**:
  + Complex reasoning → larger LLM.
  + Simple numeric optimization → smaller LLM or symbolic engine.

#### **3.4 Memory Layer**

* Stores proposals, critiques, and negotiation history.
* Must support **retrieval by round** for auditing or backtracking.

### ****4. Performance Engineering****

#### **4.1 Model Selection**

* Assign **different models** to different roles based on task complexity and cost sensitivity.
* Example: Use GPT-4 for negotiation lead, GPT-4o-mini for critique agents.
* Implement **dynamic escalation**: if agents fail to converge after N rounds, switch to a higher-capacity model.

#### **4.2 Caching**

* Cache **common proposals or evaluations** that repeat across similar negotiations.
* Example: Hotel rating criteria for multiple travel plans.

#### **4.3 Batching**

* Parallelize evaluations in the critique phase.
* Aggregate results and feed them into the next reasoning cycle in a single prompt.

### ****5. Safety and Guardrails****

* **Bias Mitigation**: Ensure agents are not favoring specific outcomes unfairly.
* **Scope Enforcement**: Agents should not access data outside their domain.
* **Round Caps**: Prevent endless debate loops.
* **Compliance Filters**: Particularly for regulated domains (finance, law, healthcare).

### ****6. Example: AI Procurement Negotiation****

**Scenario**:  
A company wants to select the best cloud provider for an AI workload.

**Agents**:

1. **Cost Agent** – Minimizes operational expenses.
2. **Performance Agent** – Optimizes for low latency and high throughput.
3. **Compliance Agent** – Ensures provider meets HIPAA and GDPR requirements.

**Negotiation Flow**:

* **Proposal Phase**:
  + Cost Agent proposes a budget-friendly option with minimal GPU cost.
  + Performance Agent suggests a high-performance tier with faster inference.
  + Compliance Agent proposes only vendors with robust audit logs.
* **Critique Phase**:
  + Cost Agent rejects expensive tier.
  + Performance Agent flags budget tier as insufficient for peak load.
  + Compliance Agent vetoes non-certified vendors.
* **Resolution Phase**:
  + Agreement on mid-tier vendor that balances cost, speed, and compliance.
* **Execution**:
  + Provisioning request sent to infra orchestration system.

### ****7. Common Pitfalls****

* **Context Explosion**: Negotiation history grows too large for context window.
* **Deadlocks**: Agents repeatedly reject each other without compromise.
* **Unequal Influence**: Stronger models dominate weaker ones, skewing outcomes.
* **Token Waste**: Repeating entire prior exchanges instead of summarizing key points.

### ****8. Key Takeaways****

* Negotiation pattern is ideal when **multiple perspectives** must converge on a single decision.
* LLD must ensure **clear agent roles, controlled communication, and stateful memory**.
* **Model selection, caching, and batching** significantly improve negotiation efficiency.
* Safety and guardrails are non-negotiable in regulated or high-stakes use cases.
* Works best when **number of rounds is capped** and **summarization techniques** keep context manageable.

### ****Unit 5****

### ****Validation & Structured Outputs****

### ****(Pydantic, JSON Schema Basics)****

### ****1. Context: Why Validation & Structured Outputs Matter in GenAI LLD****

In production-grade GenAI systems, **validation and structured outputs** are not optional—they are essential for ensuring that generated responses can be reliably consumed by downstream systems, APIs, or other agents.  
Unlike humans, software systems cannot “interpret” vague or loosely formatted answers. If a model output deviates even slightly from the expected structure, it can break pipelines, trigger incorrect actions, or cause compliance violations.

From a Low-Level Design (LLD) perspective, validation serves **three critical purposes**:

1. **Reliability** – Guarantees that outputs match the expected format, allowing automation to run without manual intervention.
2. **Compliance** – Enforces legal, ethical, or business constraints on model outputs before they reach end-users.
3. **Interoperability** – Ensures that AI outputs can be ingested by external systems (e.g., APIs, databases, other agents) without additional parsing.

In the GenAI context, two common methods for enforcing output structure are **Pydantic** (Python-centric) and **JSON Schema** (language-agnostic). Both can be combined with **guardrails** to form a robust, multi-layered safety net.

### ****2. Core Components of Validation & Structured Output Design****

#### **2.1 Pydantic for Output Validation**

Pydantic is a Python library that enforces type safety and structure validation at runtime using Python type hints and data models.

**LLD Integration Flow**:

1. Define a **BaseModel** representing the expected output schema.
2. Pass this schema to the LLM as part of the prompt (e.g., via response\_format or explicit JSON example).
3. Parse the LLM output into the Pydantic model.
4. Handle validation errors gracefully—retry generation or raise a controlled exception.

**Benefits**:

* **Type Safety**: Enforces string, int, float, list, dict types automatically.
* **Custom Validators**: Apply domain-specific constraints (e.g., rating must be 1–5).
* **Serialization**: Easily convert validated objects to JSON or dict for storage.

**Example**:

from pydantic import BaseModel, Field

from typing import List

class Restaurant(BaseModel):

name: str

cuisine: str

rating: float = Field(..., ge=0, le=5)

address: str

class RestaurantList(BaseModel):

restaurants: List[Restaurant]

#### **2.2 JSON Schema Basics**

JSON Schema is a language-agnostic way to describe JSON data structure, widely used in APIs.

**LLD Integration Flow**:

1. Define the schema in JSON format.
2. Provide the schema to the LLM in the system or developer prompt.
3. Validate the generated JSON against the schema using a JSON Schema validator.

**Benefits**:

* **Cross-Language Compatibility**: Works in multi-stack environments.
* **Clear Documentation**: Schemas double as API contracts.
* **Strict Validation**: Ensures format adherence before passing data downstream.

**Example**:

{

"type": "object",

"properties": {

"restaurants": {

"type": "array",

"items": {

"type": "object",

"properties": {

"name": {"type": "string"},

"cuisine": {"type": "string"},

"rating": {"type": "number"},

"address": {"type": "string"}

},

"required": ["name", "cuisine", "rating", "address"]

}

}

},

"required": ["restaurants"]

}

#### **2.3 Combined Approach: Pydantic + JSON Schema**

In Python-centric GenAI stacks, the most robust approach is to:

* Use **Pydantic** for runtime parsing and validation inside the service layer.
* Auto-generate **JSON Schema** from Pydantic models for external API documentation and external validation.

### ****3. Engineering Considerations in LLD****

#### **3.1 Model Selection**

* **Lightweight models** (e.g., GPT-4o-mini) may suffice for simple, static schemas.
* **Larger models** (e.g., GPT-4) are better at adhering to **complex, nested schemas**.
* Consider **model fine-tuning** for schema adherence in high-volume applications.

#### **3.2 Caching**

* Cache **validated outputs** to avoid revalidating identical requests.
* Store **validation error patterns** to optimize future prompts.

#### **3.3 Batching**

* When validating multiple outputs (e.g., processing 500 search results), batch them to reduce validation overhead.
* Combine schema validation with streaming parsers for large outputs.

### ****4. Guardrails in Validation****

* **Pre-Validation Filters**: Detect and reject outputs that are obviously invalid before schema parsing.
* **Auto-Repair Mechanisms**: If validation fails, re-prompt the model with correction instructions.
* **Field-Level Constraints**: Prevent sensitive values, enforce enumerations, or set numerical limits.

### ****5. Example: Travel Itinerary Agent with Pydantic Validation****

**User Query**: "Plan a 5-day trip to Tokyo with daily activities, cost estimates, and locations."

**Expected Output Schema**:

class Activity(BaseModel):

name: str

location: str

cost\_estimate: float

class DayPlan(BaseModel):

day: int

activities: List[Activity]

class Itinerary(BaseModel):

destination: str

days: List[DayPlan]

**Workflow**:

1. Prompt the LLM to produce output matching Itinerary.
2. Parse the LLM’s JSON output into Itinerary.
3. If parsing fails, retry generation with feedback specifying the missing or invalid fields.

### ****6. Common Pitfalls****

* **Loose Prompting**: Without explicit instructions, models output prose instead of structured JSON.
* **Overly Complex Schemas**: Deeply nested schemas increase token usage and validation errors.
* **Silent Failures**: Skipping validation on “trusted” outputs can lead to production incidents.
* **Validation Bottlenecks**: Synchronous schema checks can slow response times for high-frequency APIs.

### ****7. Key Takeaways****

* Structured output validation is essential for **automation, compliance, and interoperability** in GenAI systems.
* **Pydantic** excels in Python-native environments; **JSON Schema** ensures cross-language validation.
* Combining the two offers the most robust and flexible approach.
* **Model selection, caching, and batching** strategies directly impact validation performance and cost.
* Validation must be paired with **guardrails** for maximum reliability in production-grade LLD.

### ****Unit 6****

### ****Evaluation Layer****

### ****(RAGAS, TruLens Basics)****

### ****1. Context: Why the Evaluation Layer Matters in GenAI LLD****

In production-grade Generative AI systems, development doesn’t end when the model produces a seemingly correct answer — in fact, that’s just the starting point. The **Evaluation Layer** acts as the quality assurance and improvement engine of the system, ensuring that the model’s responses remain **accurate, relevant, safe, and compliant** over time.

Without a structured evaluation process, it’s impossible to measure whether your GenAI pipeline is meeting business objectives, maintaining user trust, or delivering ROI. Unlike traditional software, where outputs are deterministic, GenAI outputs are **probabilistic** — meaning the same query may produce slightly different results. This variability makes systematic evaluation critical at the LLD stage.

In **Low-Level Design (LLD)**, the Evaluation Layer serves three intertwined purposes:

1. **Reliability Control** – Verifies that model outputs meet predefined quality thresholds before they are delivered to the end user or another system.
2. **Feedback for Iteration** – Collects structured data that informs prompt refinement, retrieval tuning, model updates, and other iterative improvements.
3. **Compliance and Safety Monitoring** – Detects potential policy violations, hallucinations, or sensitive information leaks early in the pipeline.

Modern evaluation frameworks like **RAGAS** (for RAG-specific pipelines) and **TruLens** (for general-purpose LLM workflows) give GenAI engineers ready-to-use tools for implementing this layer efficiently.

### ****2. Core Components of an Evaluation Layer in LLD****

#### **2.1 Metric Definition & Alignment**

Defining evaluation metrics is not just a technical exercise — it’s a strategic decision that must align directly with **business goals and user expectations**. For instance, in a legal compliance assistant, “accuracy” isn’t about getting a fact roughly right — it’s about being **100% faithful** to source material and legal standards.

At the LLD level:

* **Functional Metrics** (e.g., accuracy, relevance, coverage) must measure the system’s intended purpose.
* **Operational Metrics** (e.g., latency, throughput) ensure the system meets SLAs.
* **Safety & Compliance Metrics** (e.g., toxicity, PII leakage) ensure adherence to policies and regulations.

These metrics should be **implemented as callable, testable functions** in the codebase (e.g., evaluate\_faithfulness(), check\_compliance()), making them reusable across offline and online evaluations.

#### **2.2 Tool Integration (RAGAS, TruLens)**

**RAGAS (Retrieval-Augmented Generation Assessment)**

* Tailored for RAG systems where output quality heavily depends on retrieved context.
* Evaluates:
  + **Answer Relevance** – How well the generated output addresses the original query.
  + **Context Relevance** – Whether the retrieved passages were truly helpful for answering the question.
  + **Faithfulness** – Whether the answer sticks to the retrieved context or drifts into hallucinations.
* Operates on (query, retrieved\_context, generated\_answer) triplets and produces **quantitative scores** that can be tracked over time.

**TruLens**

* A general-purpose evaluation and observability toolkit for **any LLM-based application**.
* Offers:
  + **Custom Evaluators** – Define your own scoring logic for niche use cases (e.g., "contains mandatory disclaimer").
  + **Built-in Metrics** – Readability, conciseness, relevance, and safety.
  + **Instrumentation** – Wraps LLM calls for automatic logging and scoring.
  + **Dashboards** – Enables engineers to visualize patterns, detect anomalies, and share insights across teams.

LLD teams should decide **which tool to integrate based on architecture** — RAGAS is ideal for retrieval-heavy systems, while TruLens is a better fit for multi-agent workflows or hybrid pipelines. In many cases, both can be used together.

#### **2.3 Data Capture Hooks**

The evaluation layer depends on **high-quality data capture** at every key step in the pipeline. This means:

* Logging **all user inputs** with timestamps, user metadata, and request IDs.
* Capturing **retrieved context** before it’s fed to the LLM.
* Saving the **raw model output** before post-processing or guardrail enforcement.
* Annotating each record with **model version**, **prompt template ID**, and **retrieval configuration** for traceability.

If you miss capturing any of these, post-hoc analysis becomes guesswork.

#### **2.4 Feedback Loop Design**

An evaluation layer is only valuable if its outputs are fed back into **continuous improvement cycles**:

* **Offline Batch Evaluation** – Periodic scoring of stored data to identify long-term trends and regression patterns.
* **Online Real-Time Evaluation** – Immediate scoring of each response to enable adaptive routing (e.g., send low-confidence answers for human review).
* **Auto-Optimization Triggers** – Use poor metric scores to automatically trigger prompt re-engineering, retrieval re-ranking, or model version fallback.

### ****3. LLD Implementation Details for RAGAS****

#### **3.1 Typical Setup**

1. Install the ragas package.
2. Pass query-context-answer triplets into its built-in evaluators.
3. Store the resulting scores in an **evaluation database** for tracking.

#### **3.2 Strengths**

* Optimized for **retrieval-heavy pipelines** where context quality matters as much as LLM performance.
* Gives **granular insight** into whether failures are due to poor retrieval, poor generation, or both.

#### **3.3 LLD Considerations**

* If retrieval is poor, faithfulness scores lose meaning — always ensure your retriever is performing well before relying heavily on RAGAS scores.
* Works best with **representative datasets** that mimic real-world usage patterns.

### ****4. LLD Implementation Details for TruLens****

#### **4.1 Typical Setup**

1. Install trulens\_eval.
2. Wrap your LLM client calls with TruLens instrumentation.
3. Define evaluators in Python — can be prompt-based or rule-based.
4. Enable logging to a persistent store for dashboards.

#### **4.2 Strengths**

* **Framework-Agnostic** – Works with LangChain, LlamaIndex, direct API calls, or custom agent frameworks.
* **Extensible** – Add domain-specific metrics without modifying the underlying tool.
* **Full Visibility** – Logs every step, making it easier to debug complex chains or multi-agent workflows.

#### **4.3 LLD Considerations**

* May add slight latency overhead if running complex evaluators in real time — plan for async or batched scoring where possible.
* Some evaluators themselves may require calling an LLM, adding cost.

### ****5. Engineering Considerations****

#### **5.1 Model Selection**

The evaluator model should be **fit-for-purpose**:

* Use smaller, cheaper models (e.g., GPT-4o-mini, LLaMA 3 8B) for high-volume, low-criticality evaluation tasks.
* Use larger, more capable models (e.g., GPT-4, Claude Opus) for compliance, legal, or safety-critical evaluations where nuance matters.

#### **5.2 Caching**

* Cache evaluation results for identical inputs to avoid redundant scoring.
* Maintain **historical evaluation records** to detect performance drift over weeks or months.

#### **5.3 Batching**

* Group evaluation requests into batches to reduce API overhead.
* Combine with streaming parsers for large datasets to balance speed and memory usage.

### ****6. Example: RAG-Based Policy Compliance System****

**Scenario**:  
A corporate **PolicyRAG** assistant answers compliance-related questions by retrieving internal policy documents and generating answers via an LLM.

**Evaluation Layer Flow**:

1. **Data Capture** – Log the query, top-5 retrieved documents, and generated answer.
2. **RAGAS Evaluation** –
   * answer\_relevance: Checks if the LLM’s answer directly addresses the question.
   * faithfulness: Ensures no hallucinations relative to retrieved documents.
   * context\_relevance: Confirms retrieval brought in the right policy sections.
3. **TruLens Evaluation** –
   * **Toxicity Check**: Ensures no harmful or biased language.
   * **Compliance Check**: Validates presence of required disclaimers.
4. **Feedback Loop** – Poor faithfulness scores automatically trigger a re-ranking strategy in the retrieval pipeline; repeated low scores for a prompt template flag it for rewrite.

### ****7. Common Pitfalls****

* **Metric Mismatch** – Measuring what’s easy instead of what’s meaningful; irrelevant metrics can mislead optimization efforts.
* **Over-Evaluation** – Running full evaluations on every request can balloon costs without proportional benefit.
* **Under-Sampling** – Evaluating too little data hides edge cases and rare failures.
* **Blind Trust** – Accepting evaluator scores without periodic human verification risks reinforcing systematic bias.
* **One-Off Setup** – Treating evaluation as a project milestone instead of a continuous process leads to quality decay.

### ****8. Key Takeaways****

* The Evaluation Layer is the **control center** for measuring and maintaining GenAI quality at scale.
* **RAGAS** is purpose-built for retrieval-based systems; **TruLens** is a flexible, system-wide evaluator.
* Proper **data capture, metric definition, and feedback loop design** ensure that evaluation drives real improvements.
* Cost, latency, and accuracy trade-offs should guide **evaluator model selection**.
* Continuous monitoring prevents slow quality erosion and ensures the system adapts to new challenges over time.

### ****Unit 7****

### ****Observability Layer****

### ****(LangSmith, Cost Tracking, Logging)****

### ****1. Context: Why Observability Matters in GenAI LLD****

In traditional software, **monitoring** focuses mainly on uptime, error rates, and system performance. In production-grade **Generative AI systems**, observability must extend far deeper — it’s about **understanding system behavior end-to-end** and answering not just “Is the system working?” but also “Why did it produce this specific output?”.

Generative AI pipelines are **probabilistic** and involve multiple components — LLMs, retrievers, orchestrators, guardrails, and evaluators. Without an Observability Layer, failures can be opaque, costs can spiral unnoticed, and quality drift can occur without warning.

At the **Low-Level Design (LLD)** stage, the Observability Layer is the **nervous system** of your GenAI application — capturing structured, query-level insights on **performance, quality, and cost**, and enabling engineers to **trace, debug, and optimize** workflows in real time.

In short:

* **Monitoring** tells you if something is wrong.
* **Observability** helps you figure out why it’s wrong and how to fix it.

### ****2. Core Components of the Observability Layer in LLD****

#### **2.1 Request & Response Tracing**

Every request in a GenAI system should be **traceable from entry to exit**, with visibility into each intermediate step. This includes:

* **Input capture**: Exact prompt or query text, metadata (user ID, session ID), and time of request.
* **Intermediate states**: Retrieved documents, tool calls, intermediate reasoning steps (in agent systems).
* **Output capture**: Full LLM-generated response before and after guardrails.
* **Versioning**: Model version, prompt template version, retrieval configuration ID.

This tracing allows you to reconstruct exactly what happened during an interaction — critical for debugging hallucinations or compliance violations.

#### **2.2 LangSmith Integration**

**LangSmith** (by LangChain) is a specialized observability platform for LLM-powered applications. It enables:

* **End-to-end session visualization** – See the full chain/graph execution, including prompt text, intermediate outputs, and model responses.
* **Evaluation hooks** – Attach quality metrics like relevance, faithfulness, or toxicity checks directly to the logs.
* **Version comparisons** – A/B test different prompt templates or model versions in real workloads.
* **Collaborative debugging** – Share execution traces with teammates for faster issue resolution.

In LLD terms, LangSmith can be integrated at the **middleware or orchestration layer**, so **every chain/agent call** automatically sends structured trace data to LangSmith’s backend.

**Implementation Tip**: Use LangSmith’s Python SDK or LangChain native integration to automatically log both **structured (JSON)** and **unstructured (prompt text)** data. This allows downstream analysis with filters like “show all queries that took >5 seconds” or “show all runs where context relevance < 0.7”.

#### **2.3 Cost Tracking**

LLM usage costs can scale quickly, especially when pipelines involve multiple API calls or large token contexts. A good Observability Layer must:

* **Track cost per request**: Calculate total cost at the query level, broken down by model calls.
* **Attribute cost by component**: Differentiate cost from retrieval, inference, and evaluation calls.
* **Set budget thresholds**: Trigger alerts if daily/weekly costs exceed expected budgets.
* **Compare cost-performance trade-offs**: Correlate model choice with output quality and cost efficiency.

**Engineering Detail**:

* Use model pricing tables (tokens-in/tokens-out) from vendors like OpenAI, Anthropic, or Cohere.
* Implement cost calculation utilities (e.g., calculate\_cost(tokens\_in, tokens\_out, model\_name)).
* Cache model cost constants in configuration files for easy updates.

#### **2.4 Logging Architecture**

Logs in GenAI systems must capture **structured, query-aware data**:

* **Access logs**: User request patterns, API gateway calls, authentication events.
* **Application logs**: Prompt assembly details, retrieval hits/misses, validation results.
* **LLM logs**: Raw prompts, token counts, latency, error codes.
* **Custom logs**: Domain-specific markers like “risk\_level”, “citation\_sources”, or “compliance\_passed”.

At LLD, design your logging format as **JSON** for easy parsing, indexing, and analytics (e.g., ELK Stack, Grafana Loki, Datadog).

**Implementation Tip**: Use **correlation IDs** across logs so a single request’s journey can be followed through every subsystem.

#### **2.5 Real-Time Alerts & Dashboards**

Observability is incomplete without **real-time feedback mechanisms**:

* Alerts for **latency spikes**, **retrieval failures**, **hallucination flags**, or **cost surges**.
* Dashboards showing KPIs like:
  + Average LLM latency
  + Retrieval hit rate
  + Faithfulness scores over time
  + Daily/weekly cost trends

These dashboards can be built in **Grafana, Kibana, LangSmith UI**, or integrated directly into internal admin portals.

### ****3. Engineering Considerations****

#### **3.1 Model Selection for Observability Hooks**

* Some evaluation or guardrail checks require calling an LLM (e.g., for semantic similarity scoring).
* Use **smaller, faster, cheaper models** (e.g., GPT-4o-mini, LLaMA 3 8B) for these checks to avoid inflating costs unnecessarily.

#### **3.2 Caching Observability Data**

* Cache static metadata like model pricing info or evaluation thresholds locally.
* Avoid re-logging unchanged context for repeat queries to save storage.

#### **3.3 Batching**

* Batch logging writes to avoid slowing down request handling.
* Use asynchronous queues (e.g., Kafka, RabbitMQ) for log ingestion at scale.

### ****4. Example: Observability in a PolicyRAG System****

**Scenario**:  
A corporate compliance assistant uses RAG to answer questions based on internal policy docs.

**Observability Flow**:

1. **Request Trace**: Log incoming question, user metadata, and correlation ID.
2. **LangSmith Logging**: Capture retrieval step (top-5 docs), prompt construction, LLM call, and output.
3. **Cost Tracking**: Record token usage and cost per LLM call.
4. **Alerting**: If cost per request > $0.20, send Slack alert to engineering team.
5. **Dashboard View**: See real-time graphs of retrieval hit rates, LLM latency, and cost distribution per department.

### ****5. Common Pitfalls****

* **Logging Overhead** – Excessive logging can slow the system; balance detail with performance.
* **Cost Blindness** – Tracking quality without tracking cost can lead to expensive, unsustainable pipelines.
* **Missing Correlation IDs** – Makes tracing cross-service requests extremely difficult.
* **No Actionable Alerts** – Alerts without clear remediation steps cause alert fatigue.
* **Neglecting Privacy** – Logging raw user data without redaction can create compliance risks.

### ****6. Key Takeaways****

* Observability in GenAI systems must go beyond uptime — it’s about **explainability, cost control, and traceability**.
* **LangSmith** provides purpose-built LLM observability with integrated quality evaluation.
* **Cost tracking** at the query level prevents runaway expenses and enables informed trade-offs.
* **Structured logging** with correlation IDs is non-negotiable for debugging complex pipelines.
* Observability should be **real-time, actionable, and privacy-conscious** to maintain both technical excellence and user trust.