**Module 2: HLD for GenAI**

**Unit 1**

**Component Layering in GenAI Systems**

**1. Context: Why Component Layering Matters in GenAI System Design**

Component layering is the practice of organizing a system into distinct, logically separated tiers that each serve a specific role. In traditional software systems, layered architecture improves maintainability, scalability, and clarity of system responsibilities.

In Generative AI (GenAI) systems, layering plays an even more pivotal role because these systems are inherently more complex:

* They incorporate **heterogeneous components** — LLMs, retrieval systems, data pipelines, APIs, validation services, monitoring tools.
* They must remain **flexible and evolvable** — allowing models, frameworks, or services to be swapped without wholesale redesign.
* They operate under **strict constraints** — latency, compliance, accuracy, and cost — that require precise control of responsibilities.

A layered approach delivers:

* **Separation of Concerns:** Each layer focuses on a single domain of responsibility.
* **Modularity:** Components can be replaced or upgraded without ripple effects across the entire system.
* **Better Governance:** Compliance and validation checkpoints can be placed at controlled boundaries.
* **Operational Clarity:** Engineers know exactly where to look when debugging, optimizing, or scaling.

Without well-defined layers, GenAI architectures risk devolving into tangled, tightly coupled systems — where even small changes require high-risk interventions and debugging is slow.

**2. Typical Layers in a GenAI System**

Although exact layering depends on the use case, most production-grade GenAI architectures follow a common structure of interconnected layers, each with clearly defined inputs, outputs, and responsibilities.

**1. Presentation Layer**

* The point of interaction between the user and the system — can be a web dashboard, mobile app, CLI, or even a voice interface.
* Responsible for gathering user input, presenting system outputs, and handling real-time interactions.
* Must be responsive, intuitive, and designed for minimal friction in user flows.
* Example: A chatbot interface that accepts a compliance officer’s query and returns a neatly formatted, citation-backed answer.

**2. API Gateway / Orchestration Layer**

* Serves as the **central access point** for all requests entering the system.
* Handles authentication, authorization, rate limiting, and API logging.
* For agentic or multi-model systems, this layer also orchestrates calls between tools, models, or agents.
* Example: An API endpoint that routes high-risk legal queries to a RAG-based retrieval flow, while sending FAQs directly to a lightweight LLM.

**3. Processing & Enrichment Layer**

* Transforms raw user input into structured, enriched requests suitable for LLM processing.
* Tasks include text cleaning, entity extraction, query classification, retrieval from vector databases, and context assembly.
* Example: Extracting named entities from a legal query, retrieving relevant documents from Qdrant, and compiling them into a prompt template.

**4. Model Inference Layer**

* Executes the core generation or reasoning task using one or more LLMs.
* Includes prompt templating, model selection/routing, and any pre/post-processing steps specific to the model.
* Example: Sending a structured prompt to GPT-4 for a deep legal opinion, or routing a casual clarification request to GPT-4o-mini.

**5. Validation & Guardrails Layer**

* Enforces compliance, accuracy, and formatting rules on model outputs before they reach the user.
* Implements schema validation (Pydantic), business rule checks, hallucination detection, and sensitive content filtering.
* Example: Ensuring a policy compliance answer contains mandatory fields like risk\_level and citation\_sources.

**6. Data & Integration Layer**

* Manages persistent storage, embeddings, vector search, and integration with external systems.
* May include SQL/NoSQL databases, vector databases, cloud object storage, and external API integrations.
* Example: Storing embeddings in Pinecone, logging all user queries in PostgreSQL, storing documents in AWS S3.

**7. Observability & Monitoring Layer**

* Monitors system performance, reliability, and quality of AI outputs.
* Integrates with evaluation tools such as RAGAS, TruLens, or custom dashboards.
* Example: Tracking average LLM response time, retrieval accuracy, and compliance scores in Grafana or Kibana.

**3. Design Considerations for Layering in GenAI**

When defining layers, architects must address both **engineering best practices** and **GenAI-specific operational realities**.

* **Loose Coupling:**  
  Layers must be independent enough that a change in one does not cause failures or unexpected behavior in others. This architectural principle supports rapid iteration and isolated testing, ensuring that new features or optimizations can be introduced without destabilizing the entire system. In GenAI, loose coupling is critical because model APIs, retrieval services, or compliance rules may change frequently.
* **Standard Interfaces:**  
  Clearly defined APIs, message formats, or communication contracts between layers ensure predictable interactions and make it easier to integrate new components. This also simplifies automated testing and mocking of dependencies. For example, a retrieval layer that consistently returns results in a predefined JSON schema can be swapped or upgraded without affecting downstream processing.
* **Replaceability:**  
  The architecture should allow components such as models, vector databases, or storage engines to be replaced with minimal changes to surrounding layers. This adaptability is essential in GenAI, where model vendors evolve quickly, and new frameworks or data storage solutions may offer performance or cost advantages.
* **Performance Boundaries:**  
  Every layer introduces some degree of latency or processing cost. By clearly defining the responsibilities and boundaries of each layer, you can minimize redundant operations and keep the system efficient. For example, heavy document preprocessing should occur in the enrichment layer, avoiding repeated work in the inference or validation layers.
* **Security Isolation:**  
  Layers that process sensitive or regulated data should be isolated with strict access controls, encryption in transit and at rest, and enhanced monitoring. This containment strategy reduces the surface area of potential breaches and allows compliance-focused audits to focus on a smaller, well-defined set of components.
* **Compliance Anchors:**  
  Strategic placement of validation, logging, or audit checkpoints at key boundaries ensures that governance policies are consistently enforced. In a GenAI system, this might involve PII redaction before data leaves the processing layer, or structured output validation before results are returned to the user.

**4. Benefits of a Layered GenAI Architecture**

* **Independent Scalability:**  
  Each layer can be scaled independently according to its workload, allowing fine-grained resource allocation and cost control. For example, the model inference layer might require additional GPU nodes during peak hours, while the storage layer remains constant. This avoids over-provisioning the entire system and supports different scaling strategies (vertical or horizontal) per layer.
* **Targeted Maintenance:**  
  Engineers can update, fix, or enhance a specific layer without redeploying the entire architecture. This reduces downtime, lowers risk, and shortens the release cycle. For instance, upgrading the retrieval logic in the processing layer doesn’t require changes to the presentation layer or the inference engine.
* **Experimentation Flexibility:**  
  Allows safe A/B testing of new models, retrieval strategies, or guardrail mechanisms within a single layer. Because the other layers remain unaffected, experiments can be rolled out to a subset of traffic and measured in isolation. This is critical for continuous improvement in GenAI systems, where new models and techniques emerge rapidly.
* **Failure Containment:**  
  Problems in one layer can be isolated and handled without affecting the others. For example, if the vector database in the data layer becomes temporarily unavailable, the system can fail gracefully with cached results, while the UI and inference layers remain operational. This improves resilience and user trust.
* **Regulatory Confidence:**  
  When responsibilities are clearly divided, compliance audits are easier and faster to perform. Auditors can examine only the relevant layer that handles sensitive data, such as the guardrails layer for PII masking, without needing to review the entire stack. These speeds up certification and ensures ongoing adherence to regulations like GDPR or HIPAA.
* **Enhanced Debugging:**  
  Monitoring and logs are layer-specific, enabling faster root cause analysis. Engineers can pinpoint whether an issue originates in query preprocessing, retrieval, inference, or post-processing. This targeted approach reduces mean time to resolution (MTTR) and minimizes user impact.

**5. Example: Multi-Layered PolicyRAG System**

**Scenario:** A corporate compliance assistant that answers policy-related queries using Retrieval-Augmented Generation.

* **Presentation Layer:** Web dashboard and chat interface where compliance officers input queries.
* **API Gateway / Orchestration Layer:** Handles authentication, logs query metadata, and routes requests to the correct retrieval flow.
* **Processing & Enrichment Layer:** Extracts policy terms, retrieves relevant sections from a compliance knowledge base, and assembles them into an LLM prompt.
* **Model Inference Layer:** Routes complex queries to GPT-4 for detailed analysis; directs routine checks to GPT-4o-mini for faster turnaround.
* **Validation & Guardrails Layer:** Applies JSON schema validation and removes any prohibited content before delivering the output.
* **Data & Integration Layer:** Embeddings stored in Qdrant; policy documents stored in AWS S3; integration with internal compliance tracking system.
* **Observability & Monitoring Layer:** Tracks model accuracy, average retrieval latency, and monitors compliance score metrics.

**6. Key Takeaways for AI Architects**

* Component layering in GenAI systems is foundational for achieving **scalability, modularity, and operational control**.
* Each layer must have a **clear responsibility** and communicate via standardized, well-defined interfaces.
* Some layers, such as validation and observability, are optional in traditional systems but **mandatory** in production-grade GenAI for compliance and trust.
* A layered approach allows faster iteration, safer deployments, and simpler compliance audits — enabling AI systems to adapt quickly without sacrificing reliability.

**Unit 2**

**Core GenAI Architectural Patterns**

**(RAG, Agentic, Multi-Modal)**

**1. Context: Why Core Architectural Patterns Matter in GenAI**

Generative AI systems are built on a set of recurring architectural patterns that determine how data flows, how models interact, and how outputs are validated before reaching the user. These patterns provide a blueprint for building systems that are not only functional, but also **scalable, maintainable, and adaptable** to rapidly changing AI technologies.

In traditional software, patterns such as MVC, microservices, and layered architecture have been widely used to organize code and infrastructure. In Generative AI, the patterns differ because:

* The system’s intelligence comes from **machine-learned models** rather than deterministic logic.
* Output quality depends on how models are **integrated with knowledge sources, tools, and other models**.
* Real-world deployment demands **guardrails, monitoring, and compliance** at every stage.

Three core patterns dominate the current GenAI landscape:

* **Retrieval-Augmented Generation (RAG)** – Combining LLM reasoning with external knowledge sources for factual accuracy.
* **Agentic Architecture** – Using autonomous or semi-autonomous AI “agents” to coordinate multi-step reasoning or task execution.
* **Multi-Modal Systems** – Integrating multiple data types (text, images, audio, video) for richer interactions and outputs.

Selecting and implementing the right pattern — or combination of patterns — has a direct impact on **system performance, user trust, and maintainability**.

**2. Retrieval-Augmented Generation (RAG)**

RAG is one of the most widely adopted GenAI patterns for building systems that require **grounded, factual, and up-to-date outputs**. It combines an LLM’s generative abilities with a retrieval mechanism that fetches relevant context from external data sources before generation.

**Core Components in a RAG HLD:**

* **Query Preprocessor:** Cleans and reformulates user queries for effective retrieval.
* **Retriever + Vector Database:** Finds semantically relevant documents or chunks based on embeddings.
* **Context Assembler:** Packages retrieved context into a prompt template for the LLM.
* **Model Inference Layer:** LLM generates an answer grounded in the retrieved context.
* **Validation & Guardrails:** Checks citations, ensures compliance, and reduces hallucinations.

**Design Considerations:**

* Choose vector database technology (Pinecone, Qdrant, Weaviate) based on scale and latency needs.
* Balance retrieval scope (number of chunks) with token limits.
* Implement metadata filtering to refine search results.
* Maintain an indexing pipeline for continuous data updates.

**Example:**  
A legal assistant RAG system retrieves relevant clauses from an internal policy repository and combines them with the LLM’s reasoning to answer compliance-related questions with citations.

**3. Agentic Architecture**

Agentic systems use one or more **AI “agents”** that can plan, decide, and execute tasks — often by calling tools, APIs, or other models — in a coordinated workflow. This pattern is powerful for multi-step reasoning and dynamic task decomposition.

**Core Components in an Agentic HLD:**

* **Supervisor / Orchestrator Agent:** Manages overall workflow and task assignment.
* **Specialist Agents:** Handle domain-specific subtasks (e.g., research, summarization, code execution).
* **Tooling Interface:** Connects agents to APIs, databases, or computational tools.
* **Memory Module:** Stores state across interactions for continuity and context reuse.
* **Evaluation Layer:** Validates intermediate and final outputs before delivery.

**Design Considerations:**

* Define clear communication protocols between agents (message formats, schemas).
* Manage agent autonomy to prevent runaway loops or unnecessary calls.
* Integrate observability to track decision chains for debugging and compliance.
* Use role specialization to improve efficiency and maintainability.

**Example:**  
A market research assistant where the supervisor agent delegates tasks to:

1. A research agent to gather competitor information,
2. A data analysis agent to process statistics,
3. A report-writing agent to produce a structured PDF.

**4. Multi-Modal Systems**

Multi-modal architectures allow systems to process and generate multiple types of data, enabling richer, more natural interactions and advanced capabilities.

**Core Components in a Multi-Modal HLD:**

* **Input Processing Layer:** Accepts and pre-processes various modalities (text, image, audio, video).
* **Embedding & Feature Extractors:** Converts each modality into a unified representation space.
* **Fusion Layer:** Merges features from different modalities for joint reasoning.
* **Model Inference Layer:** Uses a multi-modal LLM or a coordinated set of uni-modal models.
* **Output Generation Layer:** Produces outputs in one or more modalities.

**Design Considerations:**

* Ensure latency remains acceptable when processing heavy media (image/video).
* Decide between using a single multi-modal model (e.g., GPT-4o) or orchestrating multiple specialized models.
* Handle modality-specific compliance (e.g., biometric data from images).
* Implement caching for frequently accessed assets or embeddings.

**Example:**  
A travel planning assistant that:

* Accepts a voice query describing a trip,
* Extracts intent and entities from the audio,
* Retrieves relevant images and maps,
* Generates a textual itinerary alongside visual route maps.

**5. Choosing and Combining Patterns in HLD**

In real-world GenAI systems, patterns are often **combined**:

* A **RAG layer** can ground an **agentic system**, giving agents access to verified knowledge sources.
* A **multi-modal front-end** can feed into a **RAG pipeline**, allowing richer queries (e.g., “Here’s a photo of a product — find its compliance documentation”).

**Selection Factors:**

* **Domain Needs:** Does the domain require factual grounding, tool integration, multi-step reasoning, or multiple modalities?
* **Latency Constraints:** Agentic systems may increase latency; RAG retrieval size impacts response time.
* **Compliance Requirements:** Multi-modal inputs may add privacy risks; RAG enables better auditability.
* **Scalability Goals:** Larger agentic networks or multi-modal fusion layers require more infrastructure.

**6. Example: Hybrid GenAI System – PolicyRAG-Agentic Assistant**

**Scenario:**  
A corporate compliance assistant combines RAG and agentic patterns:

* **RAG Layer:** Retrieves up-to-date policies and regulations for grounding answers.
* **Agentic Layer:** Supervisor agent delegates tasks — one agent retrieves legal data, other checks for jurisdictional compliance, another formats final output.
* **Multi-Modal Input:** Accepts both text queries and uploaded PDF policy documents for analysis.

This hybrid approach ensures **accuracy, flexibility, and rich user interaction**.

**7. Key Takeaways for AI Architects**

* RAG, agentic, and multi-modal are the **core architectural patterns** driving modern GenAI systems.
* Each pattern addresses different needs — grounding, reasoning, and modality coverage — and can be combined for more advanced capabilities.
* Proper HLD planning ensures these patterns are **integrated cleanly** with minimal coupling, making systems adaptable to new models and frameworks.
* Choosing the right pattern(s) depends on domain requirements, latency targets, compliance considerations, and scalability goals.

### Unit 3

### Integration Patterns

### (API Gateway, Microservices, Serverless)

### 1. Context: Why Integration Patterns Matter in GenAI System Design

GenAI systems rarely function as self-contained, monolithic applications. Instead, they are **composed of interconnected components** — model inference APIs, retrieval engines, embedding generators, validation services, monitoring layers, storage systems, and presentation interfaces. The integration pattern defines how these components interact, how traffic flows between them, and how the system adapts to changes in load, models, or compliance rules.

In production-grade AI, integration patterns are not merely about connecting services; they dictate how effectively you can:

* Implement **model selection** strategies that balance cost and performance.
* Apply **caching** to reduce redundant computation.
* Use **batching** to improve throughput without sacrificing latency.
* Maintain resilience, observability, and compliance across distributed services.

Three key integration patterns are prevalent in modern GenAI HLD:

1. **API Gateway** — a unified, intelligent control point.
2. **Microservices** — loosely coupled, domain-specific services.
3. **Serverless** — event-driven execution for elastic, on-demand workloads.

### 2. Core Integration Patterns

#### **2.1 API Gateway Pattern**

**Role & Purpose in GenAI:**  
The API Gateway acts as the front door for all client requests. In traditional systems, it provides request routing, authentication, and logging. In **GenAI architectures**, it becomes an **AI-aware orchestration hub** — capable of dynamically selecting models, applying early caching, normalizing inputs, and even pre-batching requests before passing them downstream.

Consider a **multi-model compliance assistant**: The gateway can inspect incoming queries, calculate a complexity or risk score, and decide whether to send the request to a high-accuracy GPT-4 model or a cost-efficient GPT-4o-mini. This ensures **optimal resource utilization** and reduces unnecessary spend on premium models.

**Key Functions:**

* **Centralized Request Handling:** All traffic passes through a single control point, simplifying access control and routing.
* **Security Enforcement:** API key/OAuth validation, IP restrictions, rate limiting, and request throttling are applied before heavy computation.
* **Dynamic Routing:** Rules can be based on user role, query complexity, compliance sensitivity, or SLA tier.
* **Payload Normalization:** Ensures downstream services always receive data in a consistent, expected format.
* **Observability Hooks:** Logs request volume, latency, model usage patterns, and cache hit/miss ratios for performance tuning.

**Caching & Batching Intersection:**

* **Edge Caching:** The gateway can store complete responses for repeated, high-frequency queries.
* **Pre-emptive Caching:** Anticipate demand spikes (e.g., during compliance audits) and warm caches in advance.
* **Early Batching:** During peak load, group similar incoming requests to reduce duplicated retrieval or inference work.

**Design Considerations:**

* Avoid embedding complex business logic in the gateway to prevent bloated orchestration code.
* Deploy the gateway in a horizontally scalable, distributed fashion (e.g., Kong, AWS API Gateway, Envoy) to handle surges in concurrency.
* Integrate with **circuit breakers** to fail over gracefully when downstream services are unavailable.

**Common Pitfalls:**

* Over-caching without proper invalidation → stale responses in compliance-critical environments.
* Gateway as a bottleneck → overloaded orchestration logic causing latency spikes.
* Lack of regional redundancy → a single region failure can bring the system down.

#### **2.2 Microservices Pattern**

**Role & Purpose in GenAI:**  
The microservices pattern decomposes a GenAI platform into **small, independently deployable services**, each responsible for a well-defined capability. This fits naturally with GenAI workloads, as different system components have distinct performance and resource needs:

* **Retrieval service**: CPU-intensive with high I/O throughput requirements.
* **Inference service**: GPU-intensive with complex prompt-handling logic.
* **Validation service**: Lightweight but with strict compliance enforcement.

By isolating these responsibilities, each service can be **optimized, scaled, and deployed independently**. This separation makes it easier to implement advanced techniques like **model tiering**, **prompt caching**, or **vector index optimizations** without affecting unrelated parts of the system.

**Advantages in GenAI:**

* **Service Specialization:**
  + Retrieval service can fine-tune vector search algorithms, introduce ANN (approximate nearest neighbour) optimizations, and implement localized caching.
  + Inference service can manage **adaptive batching**, handle dynamic model routing, and maintain prompt result caches.
  + Guardrails service can run hallucination detection in parallel to main inference without slowing down retrieval.
* **Independent Scaling:**
  + Scale the inference service’s GPU pool during peak query hours without over-provisioning the retrieval layer.
* **Fault Isolation:**
  + Retrieval downtime can be mitigated by serving cached results while keeping inference and validation services operational.

**Caching & Batching Intersection:**

* **Local Caching:** Retrieval service caches popular query embeddings to reduce repeated vector searches.
* **Prompt Caching:** Inference service stores frequent model outputs for near-instant replay.
* **Batching:** Retrieval batches similar semantic search requests to minimize DB load; inference batches prompt executions to reduce cost per token for bulk requests.

**Design Considerations:**

* Use **contract-first API design** so changes in one service’s logic or schema don’t break others.
* Enforce strict input/output format validation for predictable inter-service behavior.
* Monitor each service independently with its own observability stack.

**Common Pitfalls:**

* Excessive network hops → increased overall latency.
* Duplicated caching strategies across services → wasted memory and risk of inconsistencies.
* Over-fragmentation into too many services → operational complexity and higher deployment overhead.

#### **2.3 Serverless Pattern**

**Role & Purpose in GenAI:**  
The serverless pattern runs workloads in **ephemeral, event-driven environments**, making it ideal for **sporadic, non-latency-critical tasks** in GenAI systems. Instead of keeping servers always running, functions execute only when triggered — perfect for workflows like document ingestion, model evaluation, or cache clean-up.

**Common GenAI Uses:**

* **Automated Document Ingestion:** On file upload, trigger a workflow to chunk, embed, and index content in a vector database.
* **Periodic Model Evaluation:** Scheduled batch testing with RAGAS/TruLens to track model drift or accuracy degradation.
* **Cache Maintenance:** Automated clean-up of expired entries to avoid serving outdated responses.

**Advantages:**

* **Elastic Scaling:** Handles unpredictable workloads without pre-provisioning.
* **Cost Efficiency:** Pay only for execution time, avoiding idle infrastructure costs.
* **Low Ops Overhead:** Cloud provider manages patching, scaling, and basic monitoring.

**Caching & Batching Intersection:**

* **Cache Pre-Warming:** Functions can populate caches before expected peak usage.
* **Bulk Processing:** During ingestion, batch document embeddings and index them in bulk to minimize per-document cost.
* **Event-Driven Cache Invalidation:** Trigger cache purges automatically when new data is added.

**Design Considerations:**

* Stay within execution time limits (commonly 5–15 minutes) to prevent timeouts.
* Use orchestrators (e.g., AWS Step Functions) for multi-step serverless workflows.
* For GPU workloads, consider container-based serverless (AWS Fargate) to avoid long GPU cold starts.

**Common Pitfalls:**

* **Cold Start Latency:** Can add significant delay if used on a real-time request path.
* **Stateless Execution:** Requires external storage to maintain state across executions.
* **Distributed Debugging:** Requires strong observability and tracing to diagnose issues across functions.

### 3. Design Considerations Across Patterns

* **Model Selection Logic:**
  + API Gateway → Best for early routing based on request metadata.
  + Inference Microservice → Best for in-flight selection based on live response metrics.
* **Caching Strategy Placement:**
  + Edge caches at the gateway for high-frequency queries.
  + Computation-heavy caches (embeddings, prompt outputs) inside the service that owns the computation.
* **Batching Scope:**
  + Gateway-level batching for external-facing workloads.
  + Service-level batching for internal processing efficiency.
* **Security & Compliance:**
  + Encrypt sensitive cached data and control access.
  + Isolate compliance-heavy workflows in separate microservices or serverless functions.

### 4. Example Scenario – Multi-Pattern PolicyRAG Deployment

**Use Case:** A corporate compliance assistant that processes policy-related queries.

* **API Gateway:**
  + Routes complex, high-risk queries to RAG + GPT-4 and low-complexity queries to GPT-4o-mini.
  + Maintains an edge cache of common compliance questions.
* **Microservices:**
  + Retrieval Service: Caches vector search results with 15-minute TTL.
  + Inference Service: Uses prompt caching and adaptive batching for similar compliance checks.
  + Validation Service: Enforces JSON schema compliance and removes prohibited terms.
* **Serverless:**
  + Automates ingestion of updated policy documents, batching embedding generation before indexing in Qdrant.
  + Performs nightly clean-up of stale cache entries to maintain compliance.

### 5. Key Takeaways

* Integration patterns define the **flow, resilience, and scalability** of a GenAI system.
* API Gateways serve as **intelligent entry points** for model selection, caching, and batching.
* Microservices allow **service-specific optimization** without affecting unrelated components.
* Serverless offers **elastic, event-driven processing** for non-real-time workflows.
* Most production GenAI systems use **a hybrid of these patterns** to balance performance, cost, and maintainability.

### Unit 4

### Security, Compliance, and Guardrails (at HLD Level)

### 1. Context: Why Security, Compliance, and Guardrails Matter in GenAI HLD

In traditional system design, security and compliance are often layered in after functionality is built. In **production-grade GenAI systems**, this approach is dangerous and costly. The nature of LLMs, retrieval systems, and agentic orchestration introduces **unique risks**:

* Sensitive or personally identifiable information (PII) might leak through prompts or outputs.
* Models may produce hallucinations that lead to incorrect or non-compliant decisions.
* Compliance requirements may span multiple jurisdictions (GDPR, HIPAA, DPDP, SOC 2), requiring **regional data segregation** and auditable workflows.

At the **High-Level Design (HLD)** stage, architects must build **security and compliance guardrails into the core architecture**, not treat them as optional add-ons. The challenge lies in balancing **model performance, caching efficiency, and batching throughput** with **robust enforcement of trust boundaries**.

### 2. Core Components of Security, Compliance, and Guardrails in GenAI HLD

#### **2.1 Secure API Gateway and Request Ingress**

The API Gateway is the first enforcement point in the architecture and plays a critical role in **access control, input sanitization, and early-stage compliance checks**.

**Responsibilities:**

* **Authentication & Authorization:** Enforce OAuth 2.0, JWT tokens, or mutual TLS.
* **Input Validation:** Strip or mask sensitive identifiers before they enter deeper layers.
* **Rate Limiting & Abuse Prevention:** Throttle excessive calls to protect downstream services and maintain SLA commitments.
* **Request Classification for Compliance Routing:**
  + High-sensitivity queries → Route to additional validation before inference.
  + Low-sensitivity queries → Route to standard processing path.

**Model Selection & Caching Impact:**

* API Gateway can select safe, compliant model endpoints for certain query classes (e.g., HIPAA-certified LLM endpoints for healthcare data).
* Caching at the edge must be compliance-aware — ensuring PII-containing results are **never cached** or are cached with encryption and strict TTLs.

**Pitfalls:**

* Overlooking the need for **multi-region API gateways** in global deployments, leading to compliance breaches when data crosses borders.
* Caching sensitive outputs without encryption, exposing them to unauthorized access.

#### **2.2 Data Governance & Compliance Anchors**

Data governance defines **what data is collected, where it is stored, how it is processed, and who can access it**. At HLD, architects must define **compliance anchor points** — specific boundaries in the data flow where validation, logging, and audits are enforced.

**Key Practices:**

* **Data Classification Layer:** Tag data at ingestion (e.g., PII, confidential, public) to drive downstream handling.
* **Jurisdictional Storage Control:** Keep EU data in EU-hosted databases to meet GDPR requirements.
* **Immutable Audit Trails:** Log all interactions, retrievals, and model outputs with tamper-proof storage.
* **PII Redaction Pre-Processing:** Remove personal identifiers before retrieval or inference to minimize leakage risk.

**Caching & Batching Impact:**

* Compliance-aware batching ensures that grouped requests don’t mix data with incompatible jurisdictional requirements.
* Distributed caching solutions must be geo-aware to store data only in approved regions.

**Pitfalls:**

* Failing to maintain an audit trail that is both **complete** and **comprehensible** to auditors.
* Hardcoding compliance rules, making them inflexible when laws change.

#### **2.3 Model-Level Guardrails**

Model-level guardrails ensure that **outputs are accurate, safe, and compliant** before being delivered to the end user. At HLD, this involves deciding **where and how** validation layers are integrated into the architecture.

**Guardrail Types:**

* **Content Filters:** Remove toxic or prohibited terms from outputs.
* **Schema Validation:** Use Pydantic or Guardrails.ai to enforce structured, predictable responses.
* **Factual Verification:** Cross-check outputs against trusted sources or retrieval context to reduce hallucinations.
* **Policy Enforcement:** Ensure outputs adhere to industry-specific rules (e.g., no investment advice disclaimers missing in finance).

**Model Selection & Caching Impact:**

* Certain high-risk outputs may bypass caching to avoid stale or incorrect information being reused.
* Guardrails can determine model selection — routing risky requests to models with better safety fine-tuning.

**Pitfalls:**

* Adding guardrails that significantly increase latency due to heavy post-processing.
* Overly aggressive filters that censor legitimate responses.

#### **2.4 Observability and Real-Time Incident Response**

At HLD, security is incomplete without **continuous observability** and a plan for rapid incident containment.

**Key Practices:**

* **Real-Time Monitoring:** Track abnormal query patterns, model output anomalies, and failed compliance checks.
* **Alerting Pipelines:** Automated escalation when thresholds are breached (e.g., sudden spike in toxic content detections).
* **Forensic Logging:** Detailed capture of full request-response cycles for post-incident analysis.

**Batching & Caching Impact:**

* Monitor batch processing jobs for compliance breaches at the group level, not just per request.
* Observe cache hit rates and validate that sensitive data is not unintentionally being served from cache.

**Pitfalls:**

* Alert fatigue from overly sensitive thresholds.
* No clear escalation path when a compliance breach is detected.

### 3. Design Considerations at HLD Level

* **Security at Entry Points:** Enforce zero-trust at API Gateway and ingress points.
* **Compliance by Design:** Architect for multi-region compliance from the start.
* **Guardrail Placement:** Balance safety enforcement between pre-inference (for data input) and post-inference (for model outputs).
* **Caching Strategy:** Ensure compliance-aware cache layers; apply encryption and TTL for sensitive data.
* **Batch Processing Boundaries:** Prevent mixing of requests from different compliance domains in the same batch.

### 4. Example Scenario – Healthcare GenAI Assistant

**Use Case:** A healthcare provider deploys a GenAI assistant to help doctors query patient records and medical guidelines.

**Implementation Highlights:**

* **API Gateway:**
  + Validates physician credentials via OAuth 2.0.
  + Filters requests to remove patient identifiers before retrieval.
  + Routes any query mentioning patient data to a HIPAA-compliant LLM endpoint.
* **Data Governance Anchors:**
  + Stores U.S. patient data only in U.S.-based HIPAA-certified databases.
  + Logs all access events to immutable storage for compliance audits.
* **Model Guardrails:**
  + Enforces JSON schema requiring a source\_references field for every medical recommendation.
  + Filters outputs to block any unsupported medical claims.
* **Observability:**
  + Monitors all generated advice for compliance with medical safety rules.
  + Triggers alerts if any output contains prohibited or unverified treatment recommendations.

### 5. Key Takeaways

* Security, compliance, and guardrails are **core architectural concerns** in GenAI HLD, not post-build patches.
* API Gateways serve as **the first line of defence**, capable of routing, filtering, and enforcing compliance-aware caching.
* Data governance requires **clear compliance anchor points** and geo-specific storage controls.
* Model-level guardrails reduce hallucinations, enforce policies, and ensure safe outputs.
* Observability and incident response must be planned from day one to ensure resilience and trust.

## 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.