**Module 1: Foundations**

**Unit 1**

**Evolution of System Design in AI**

**1. Context: Why Study the Evolution of System Design in AI?**

Understanding the evolution of system design in AI isn’t just about history—it’s about why modern AI architectures look the way they do and where they’re heading.

In traditional software, system design was centered around deterministic logic: inputs + rules = predictable outputs.

In AI, especially Generative AI, the paradigm shifted: we now design probabilistic, adaptive, and data-driven systems where outputs emerge from patterns in massive datasets, not just from explicitly coded rules.

Studying the progression helps us:

* Recognize patterns in architectural decisions over decades.
* Appreciate trade-offs that shaped today’s frameworks.
* Predict future design trends to stay ahead.

**2. Phase 1 – Pre-AI & Early AI Era (1950s–1990s)**

**Rule-Based & Expert Systems**  
Design approach was monolithic applications with explicit logic encoded by domain experts.

Examples:

* MYCIN (medical diagnosis, 1970s) – decision trees and IF-THEN rules.
* DENDRAL (chemical analysis, 1960s).

System design traits:

* Centralized control.
* Tight coupling between data, rules, and inference engine.
* Minimal scalability concerns—systems were built for single-node environments.

Limitations:

* Brittle: adding new knowledge often required redesigning core logic.
* No capacity to learn from data—everything was hardcoded.

**Design Pattern:** Pipeline of Rules  
Input Parser → Rule Matching Engine → Output Generator.  
Entire logic predefined; no stochastic components.

**3. Phase 2 – Statistical & Machine Learning Era (2000s–2015)**

**Shift from Rules to Models**  
Systems started ingesting real-world datasets and learning patterns rather than relying solely on expert-encoded rules.

ML models introduced modularity:

* Data ingestion
* Feature engineering
* Model training
* Prediction APIs

Examples:

* Search engines with ranking models.
* Recommendation systems at Netflix/Amazon.

System design traits:

* Separation of data pipelines from model serving layers.
* Microservice adoption began for scalability.
* Batch inference dominated—real-time inference was rare.

Challenges introduced:

* Need for data versioning.
* Retraining pipelines to adapt to concept drift.

**Design Pattern:** Model-Centric Architecture  
Data Pipeline → Model Training → Model Deployment → Model API → Client Applications.

**4. Phase 3 – Deep Learning Era (2015–2020)**

**Rise of Neural Networks**  
Explosion of compute power (GPUs/TPUs) enabled complex models like CNNs, RNNs, and Transformers.

System design evolution:

* Training moved to distributed, multi-node setups.
* Inference required GPU-powered serving layers.
* Data pipelines became streaming-oriented for near real-time predictions.

Examples:

* Computer vision pipelines in self-driving cars.
* NLP pipelines with BERT, GPT-2.

Architectural traits:

* Two-tier ML systems: heavy training pipelines (offline) and lightweight, optimized inference services (online).
* Model compression, quantization, and batching for performance.

Shift in responsibility:

* Engineers now needed ML + DevOps skills → MLOps emerged.

**5. Phase 4 – Generative AI Era (2020–Present)**

**Large Language Models (LLMs) & Foundation Models**  
Introduction of GPT-3, PaLM, Claude, Gemini, and others changed system design priorities:

* From model training to model integration — many teams now consume API-based LLMs instead of training from scratch.
* Systems must handle prompt engineering, context management, tool integration, and guardrails.

Architectural evolution:

* LLMOps replaces traditional MLOps in some contexts.
* Orchestration frameworks (LangChain, LangGraph, AutoGen, CrewAI) introduced agentic workflows.
* Retrieval-Augmented Generation (RAG) pipelines integrate vector databases for knowledge grounding.

New challenges:

* Latency vs. accuracy trade-offs for multi-turn conversations.
* Data privacy, compliance, and hallucination prevention.
* Multi-LLM routing for cost and capability optimization.

**Design Pattern:** LLM-Orchestrated Systems  
User Query → Preprocessing & Context Enrichment → LLM Invocation → Postprocessing → Guardrails → Response Delivery.  
Agents coordinate sub-tasks such as retrieval, calculation, summarization, and API calls.

**6. Phase 5 – The Future of AI System Design (2025 → )**

Trends emerging:

* Hybrid architectures: AI + deterministic rules for compliance-critical domains.
* On-device inference: edge deployments for latency and privacy.
* Multi-agent collaboration: autonomous systems coordinating tasks.
* Self-healing pipelines: automated retraining, evaluation, and redeployment.
* Ethics-first design: built-in bias detection and explainability layers.

**7. Comparisons Across Phases**

| **Era** | **Core Driver** | **Design Style** | **Key Limitation** |
| --- | --- | --- | --- |
| Rule-Based | Expert Rules | Monolithic | No learning |
| ML Era | Data Patterns | Modular | Retraining complexity |
| Deep Learning | Compute + Data | Distributed | Resource heavy |
| Generative AI | Foundation Models | Orchestrated + Agentic | Hallucinations, high cost |

**8. Key Takeaways for Modern AI Architects**

* Patterns repeat: each era solves previous limitations but introduces new ones.
* Integration skills matter: modern AI architects often integrate, orchestrate, and govern pre-trained models rather than build from scratch.
* System design is now socio-technical: you must design for accuracy, ethics, scalability, and maintainability simultaneously.
* Generative AI systems demand orchestration: think in terms of pipelines + agents + guardrails instead of monolithic LLM calls.

### Unit 2

**Core Principles of System Design**

**(Scalability, Latency, Trade-offs)**

### 1. Context: Why These Principles Matter in AI System Design

In Generative AI systems, design decisions have far-reaching consequences on performance, cost, reliability, and user experience. Traditional applications often operate in predictable environments, where the number of transactions and type of workloads can be estimated with reasonable accuracy. AI workloads, however, are inherently unpredictable. A user request could be a short fact-check or a complex multi-step reasoning task that requires multiple LLM calls, retrieval operations, and even coordination between different agents.

This unpredictability amplifies the importance of three core system design principles:

* **Scalability** – the system’s ability to gracefully handle increasing workloads without loss of performance.
* **Latency** – the responsiveness of the system in delivering results to the user.
* **Trade-offs** – the conscious choices made to balance competing demands such as speed versus accuracy, or cost versus scalability.

By deeply understanding and applying these principles, architects ensure that AI systems can grow sustainably, adapt to demand spikes, and deliver consistent, high-quality results across varying scenarios.

### 2. Scalability

Scalability determines how well a system can expand its capacity to meet growing demand. In AI, this goes beyond handling more concurrent users — it includes managing resource-heavy inference calls, scaling vector database queries to billions of entries, and maintaining performance for complex orchestration workflows.

**Types of Scaling:**

* **Vertical Scaling (Scale-Up)**  
  This approach increases the resources of a single machine, such as upgrading CPUs, adding RAM, or replacing a GPU with a more powerful one.  
  Pros: Straightforward to implement; minimal software changes required.  
  Cons: Hardware upgrade limits are quickly reached; single point of failure remains.  
  Example: Upgrading a GPU from 16GB to 48GB to accommodate larger LLM context sizes and more tokens per request.
* **Horizontal Scaling (Scale-Out)**  
  This method adds more machines or instances to share the workload.  
  Pros: Allows virtually unlimited scaling potential; improves fault tolerance through redundancy.  
  Cons: Requires effective load balancing, data distribution, and state management across nodes.  
  Example: Deploying a Kubernetes cluster that dynamically adds inference pods during traffic surges to maintain response times.

**Scaling in AI Systems:**

* Model serving through GPU clusters with intelligent autoscaling rules to handle peak demand.
* Distributed data processing pipelines (Spark, Ray, Dask) for large-scale training and preprocessing.
* Sharded vector databases (Pinecone, Qdrant, Weaviate) to perform large-scale semantic searches in parallel.

**Key Considerations:**

* Design services to be stateless so they can be scaled horizontally without complex state replication.
* Cache high-frequency query results to avoid reprocessing identical inputs.
* Use asynchronous job queues to process heavy tasks in the background without blocking main services.
* Plan for resource monitoring and cost controls as scaling can rapidly increase operational expenses.

### 3. Latency

Latency is the total time from receiving a user’s request to delivering a complete response. In Generative AI, latency is the sum of multiple factors — network transfer times, vector store retrieval, LLM inference, and any post-processing or formatting. Even a small delay at each stage can combine to noticeably impact the user experience.

**Types of Latency:**

* **End-to-End Latency** – The complete round trip from request initiation to final response delivery, covering all processing steps.
* **Inference Latency** – The time taken by the AI model to generate its output after receiving the prompt. For LLMs, this is heavily influenced by model size, token length, and hardware capabilities.
* **Retrieval Latency** – The time needed to locate and fetch relevant data from a database or vector store before the LLM call.

**Latency Sensitivity Across Use Cases:**

* Conversational assistants – Should respond in under 1 second for smooth, human-like interaction. Streaming partial results often helps meet this target.
* Document summarization – Can tolerate delays of 5–10 seconds if they result in higher-quality, more accurate summaries.
* Real-time fraud detection – Requires responses in milliseconds to prevent financial loss or security breaches.

**Reducing Latency in AI Systems:**

* Optimize prompts to reduce token count while maintaining essential context.
* Reduce the number of retrieved chunks in RAG pipelines to limit LLM input size.
* Deploy inference services in multiple geographic regions to minimize network latency.
* Use smaller, distilled models for simpler queries and reserve larger models for complex ones.
* Pre-compute and cache embeddings for documents that are queried frequently.
* Stream tokens to begin showing results before the full output is generated.

### 4. Trade-offs

Every AI system involves trade-offs because resources, time, and budgets are finite. The challenge is that improving one dimension often comes at the cost of another. In AI systems, these trade-offs are especially visible in model selection, retrieval design, and performance tuning.

**Common Trade-offs:**

* **Accuracy vs. Latency**  
  Larger models (e.g., GPT-4) typically deliver more accurate and nuanced responses but take longer to process, increasing latency.  
  Solution: Implement model routing logic — use smaller models for routine queries and larger models for complex or high-risk requests.
* **Cost vs. Scalability**  
  Scaling an AI service to handle thousands of concurrent LLM calls can lead to high infrastructure and API costs.  
  Solution: Cache responses for repeated queries, fine-tune smaller models for frequent use cases, and optimize batch processing where applicable.
* **Recall vs. Precision in Retrieval**  
  Expanding the search scope increases recall but can result in less relevant context, raising token costs and hallucination risks.  
  Solution: Use hybrid retrieval strategies combining semantic search with metadata filters to balance both.

**Generative AI–Specific Factors:**

* Prompt length versus token usage cost.
* Streaming output speed versus maintaining sentence coherence.
* Fine-tuning accuracy improvements versus the expense and complexity of retraining.

### 5. Example: Compliance RAG Assistant

**Scenario:** An AI chatbot assists compliance officers by answering queries based on internal company policies.

* **Scalability:** Horizontal scaling for both GPU-based inference servers and sharded vector database clusters to ensure consistent performance during peak usage.
* **Latency:** Pre-compute embeddings for frequently referenced policy sections and reduce retrieval from 10 chunks to 5 to minimize input size to the LLM.
* **Trade-offs:**
  + Use GPT-4 for nuanced legal interpretations (higher accuracy, slower responses).
  + Use GPT-4o-mini for common and straightforward queries (faster, lower cost).

### 6. Key Takeaways for AI Architects

* Scalability ensures the system can handle both gradual growth and sudden spikes without service degradation — vertical scaling for quick gains, horizontal scaling for long-term stability.
* Latency must be optimized across every stage of the pipeline, from retrieval to inference to response delivery, as delays compound quickly in AI systems.
* Trade-offs are unavoidable — make them explicit, document the reasoning, and ensure they align with business and user priorities.
* In Generative AI systems, scalability, latency, and trade-offs are interlinked; decisions in one area can directly impact the others, requiring a holistic design approach.

### Unit 3

**GenAI-Specific Constraints**

**(Latency, Token Limits, Compliance)**

### 1. Context: Why These Constraints Are Critical in Generative AI Systems

Generative AI systems face operational boundaries that go beyond traditional engineering limits. While general system design addresses scalability, fault tolerance, and performance, Generative AI introduces unique constraints tied to the behavior and infrastructure of large language models (LLMs) and other generative models.

Three constraints in particular directly influence architecture and user experience:

* **Latency** – the time taken for the model to generate a response, influenced by multiple AI-specific factors.
* **Token Limits** – restrictions on the amount of input and output data the model can process in a single request.
* **Compliance** – adherence to data privacy, legal, and industry-specific regulations, often heightened due to the nature of AI outputs.

Ignoring these constraints can lead to degraded user experience, higher costs, and regulatory risks. Incorporating them into system design from the outset is essential for building robust and trustworthy AI applications.

### 2. Latency in Generative AI

Latency in Generative AI is influenced by a combination of infrastructure, model architecture, and content generation complexity. Unlike standard APIs that return results in a fixed time window, LLMs process requests token-by-token, making performance less predictable.

**Sources of Latency:**

* **Inference Time:** The speed at which the model generates tokens depends on model size, number of parameters, and hardware (e.g., GPU/TPU performance).
* **Prompt Complexity:** Longer prompts and more retrieval context increase processing time.
* **External Calls:** RAG pipelines add retrieval latency before the LLM call, especially when querying large vector stores.
* **Streaming vs. Batch Output:** Streaming can reduce perceived latency, while batch output waits for completion before showing results.

**Design Considerations:**

* Use smaller or distilled models for low-complexity queries to improve response time.
* Stream responses to the user for conversational applications.
* Optimize prompt templates and retrieval size to reduce unnecessary tokens.
* Deploy inference endpoints in multiple geographic locations for proximity-based routing.

### 3. Token Limits

Token limits define the maximum number of tokens (input + output) that can be processed in a single LLM call. These limits vary across models and directly impact application design.

**Implications of Token Limits:**

* **Context Truncation:** When prompts exceed the limit, older context may be dropped, affecting coherence in multi-turn conversations.
* **Retrieval Boundaries:** In RAG systems, the number of chunks retrieved must be balanced to avoid exceeding limits.
* **Output Restrictions:** Long-form generation (e.g., detailed reports) may need chunked output strategies.

**Design Considerations:**

* Monitor token usage in real time and implement guards to prevent exceeding limits.
* Use summarization or context distillation for older conversation history.
* Implement sliding window or rolling context strategies for chatbots.
* Pre-filter and rank retrieved chunks before including them in the prompt.

### 4. Compliance

Compliance in Generative AI extends beyond standard data protection to include model-specific risks like hallucinations, biased outputs, and regulatory violations. AI systems must comply with frameworks such as GDPR, HIPAA, or local data residency laws, depending on the domain and geography.

**Key Compliance Risks:**

* **Data Privacy:** Ensuring personally identifiable information (PII) is not stored, leaked, or exposed in prompts or outputs.
* **Domain Regulations:** Healthcare, finance, and legal applications have strict rules on data use and retention.
* **Content Safety:** Preventing harmful, discriminatory, or non-compliant language generation.
* **Auditability:** Ability to trace responses back to source data or reasoning steps for verification.

**Design Considerations:**

* Use input and output filters for PII detection and masking.
* Store only anonymized or aggregated data in logs.
* Implement guardrails to prevent generation of unsafe or prohibited content.
* Maintain detailed audit logs for compliance verification and incident investigation.

### 5. Example: Healthcare GenAI Assistant

**Scenario:** An AI assistant provides clinicians with patient case summaries using a RAG pipeline.

* **Latency:** Streams initial summary sentences while the remainder is being generated to keep interaction fluid.
* **Token Limits:** Uses a context distillation step to summarize multiple retrieved chunks into a compact input for the model.
* **Compliance:** All patient identifiers are masked before retrieval or model input; audit logs are stored with pseudonymized references only.

### 6. Key Takeaways for AI Architects

* Latency in GenAI is affected by both model-level and system-level factors — optimize prompts, retrieval, and deployment locations to minimize delays.
* Token limits require deliberate context management strategies, especially for multi-turn conversations and large document processing.
* Compliance is not optional — design systems with built-in privacy protection, content safety, and auditability from the start.
* These constraints are interconnected: reducing token usage can lower latency, and compliance measures often influence both design and performance.

### Unit 4

**HLD vs LLD in GenAI**

### 1. Context: Why HLD and LLD Matter in Generative AI System Design

High-Level Design (HLD) and Low-Level Design (LLD) are essential stages in system architecture, translating a concept into an actionable, buildable system. In traditional software engineering, they define different layers of abstraction. In Generative AI, these concepts extend further — designs must account for AI-specific challenges such as model orchestration, data retrieval strategies, validation layers, and compliance requirements.

Understanding the differences and connections between HLD and LLD in GenAI ensures that systems are not only functionally correct but also efficient, scalable, and safe. Skipping one or blurring the distinction often leads to architecture drift, costly redesigns, and increased technical debt.

### 2. High-Level Design (HLD) in Generative AI

HLD provides the **big-picture view** of the system. It captures **what** components exist, **how** they interact, and **why** each is included — without detailing the internal implementation.

**Focus Areas in GenAI HLD:**

* **Core Components:** LLM selection, retrieval layer, orchestration framework, API gateways, UI/UX entry points.
* **Integration Points:** External APIs, vector databases, cloud services, monitoring systems.
* **Data Flow:** From user input to response delivery, showing preprocessing, retrieval, inference, and post-processing stages.
* **Security and Compliance Boundaries:** Where data enters/leaves the system, encryption, and filtering points.
* **Scaling Strategy:** Vertical vs. horizontal scaling plans for inference servers and storage systems.

**Outputs of HLD:**

* Architecture diagrams (component-level)
* Technology stack decisions
* Interaction flows between major subsystems
* Non-functional requirements (latency targets, availability, compliance constraints)

**Example in GenAI:**  
For a RAG-based medical assistant, the HLD might show a flow:  
User → API Gateway → Query Preprocessor → Retriever + Vector DB → LLM → Compliance Guardrails → Response Delivery.

### 3. Low-Level Design (LLD) in Generative AI

LLD translates the high-level architecture into **detailed implementation plans**. It specifies **how** each module in the HLD will be built, configured, and integrated.

**Focus Areas in GenAI LLD:**

* **Class and Module Design:** Detailed descriptions of agents, services, and utilities.
* **Data Structures:** Schema for document metadata, prompt templates, and structured outputs.
* **Algorithm Details:** How retrieval ranking is calculated, how model routing works, how guardrails are applied.
* **API Specifications:** Endpoints, request/response formats, authentication mechanisms.
* **Validation Logic:** Pydantic models, schema checks, field validators.
* **Error Handling:** Retry policies for failed LLM calls, fallback strategies for unavailable models.

**Outputs of LLD:**

* Sequence diagrams for multi-step flows (e.g., agent interactions).
* Detailed configuration files and environment setup instructions.
* Module-level pseudocode or partial implementations.
* Logging and observability integration plans.

**Example in GenAI:**  
For the same RAG medical assistant, the LLD might define:

* Class MedicalRetriever with methods for filtering patient data.
* Pydantic schema PatientCaseSummary for structured output.
* Prompt template structure with placeholders for retrieved context.
* Retry logic for LLM API calls with exponential backoff.

### 4. Key Differences Between HLD and LLD in GenAI

| **Aspect** | **HLD** | **LLD** |
| --- | --- | --- |
| **Abstraction Level** | High-level overview | Detailed implementation |
| **Primary Audience** | Architects, senior engineers, stakeholders | Developers, QA, DevOps |
| **Focus** | Components, interactions, data flow | Classes, methods,  APIs, algorithms |
| **Output** | Architecture diagrams, stack selection, high-level flows | Sequence diagrams, schemas, config files |
| **Timeframe** | Early design phase | Post-HLD, before coding starts |

### 5. Interplay Between HLD and LLD in GenAI Projects

In a Generative AI project, HLD and LLD are not isolated — they form a continuous pipeline from concept to code. A well-prepared HLD reduces ambiguity in LLD, while a detailed LLD ensures the HLD vision is faithfully implemented.

* **HLD informs LLD:** Technology choices in HLD dictate the implementation details in LLD (e.g., selecting LangChain in HLD means LLD will define LangChain agents, chains, and memory modules).
* **LLD refines HLD:** While working on LLD, developers may discover practical constraints that require adjusting the HLD (e.g., changing model selection strategy due to latency issues).

This interplay is particularly important in GenAI, where rapid model evolution and changing API capabilities can impact both levels of design.

### 6. Example: PolicyRAG-SecGPT Project

**HLD Elements:**

* Diagram showing ingestion pipeline, vector DB, retrieval service, LLM orchestration layer, guardrails, and API gateway.
* Technology stack: OpenAI GPT-4, Qdrant, LangChain, Guardrails.ai, AWS Lambda.

**LLD Elements:**

* Detailed schema for storing document chunks with jurisdiction metadata.
* Pydantic model for compliance answers with risk\_level, citation\_sources, and recommendations.
* API endpoint /compliance/query with request/response JSON formats.
* Error handling with retry and fallback to a smaller local model when API quota is exhausted.

### 7. Key Takeaways for AI Architects

* HLD defines **what** the system will look like; LLD defines **how** it will be built.
* Both are critical in GenAI due to rapidly evolving frameworks, strict compliance needs, and performance constraints.
* Maintain versioned documentation for both HLD and LLD to track architectural evolution alongside model changes.
* In complex agentic systems, clarity in HLD prevents confusion during LLD, reducing integration issues and rework.