# INTRODUCTION

### **Introduction**

In a world increasingly driven by information technologies, the effective and secure management of data has become a major concern for organizations. The growing volume of information generated daily requires advanced IT solutions capable of centralizing, structuring, protecting, and efficiently utilizing this data. It is within this context that **Computer-Assisted Data Management (CADM)** emerges—a discipline that combines modern digital tools with rigorous methods of data modeling and organization.

CADM represents a strategic field that not only enables the automation of data collection and processing, but also significantly enhances decision-making through well-designed databases and high-performing information systems. Through this approach, it becomes possible to ensure the integrity, security, and traceability of data across various sectors, whether industrial, administrative, or scientific.

This undergraduate dissertation is rooted in this context and aims to explore the theoretical foundations, data models, and technological tools underlying CADM platforms. Through a structured methodological approach, we will seek to design a reliable data management architecture, with a particular focus on modeling, security, and the integration of digital tools.

### **Problem Statement**

In the era of digital transformation, data management has become a fundamental pillar of organizational performance. However, it faces several major challenges: the growing complexity of information flows, the diversity of data sources, increasing security requirements, and the need for coherent and automated structuring. How, then, can we design a **Computer-Assisted Data Management (CADM)** platform capable of meeting these demands? What theoretical approaches, technological tools, and data models should be deployed to ensure efficiency, security, and scalability in data management?

### **Proposed Plan**

* **Chapter 1**: General Concepts of Data Management
* **Chapter 2**:
* **Chapitre03:**

### **Objectives of the Thesis**

This undergraduate dissertation aims to design a **Computer-Assisted Data Management (CADM)** platform that ensures efficient organization and security of information. It seeks to explore the theoretical foundations, technological tools, and data models required to implement a reliable, secure, and scalable system.

The specific objectives are:

* To understand the basic principles of CADM.
* To identify appropriate tools and technologies.
* To design a functional system architecture.
* To implement a secure illustrative prototype.
* To evaluate the results and propose avenues for improvement.

### **Conclusion**

This undergraduate dissertation is thus part of an effort to deepen the understanding of computer-based data management systems through the study of CADM. By combining solid theoretical foundations with a practical implementation, it will offer a relevant solution for the structured, secure, and scalable management of information. The following work will detail the concepts, tools, and methods required for the design of such a platform, while highlighting the issues related to data quality, security, and accessibility.

# Historical Evolution and Core Concepts of Data Management and Security

### Historical Development:

The evolution of data management and security has progressed through four major eras. During the Mainframe Era (1960s–1970s), data systems were built around mainframes using hierarchical and network models. Security relied on physical controls and simple authentication. Edgar Codd's relational model introduced structure and logic, forming a foundation for complex data access control.

In the Commercial Expansion (1980s–1990s), relational databases became mainstream. This brought heightened concerns about network vulnerabilities, prompting the development of role-based access control, encryption standards, and the use of firewalls in response to growing PC and internet use.

The Internet Age (2000s–2010s) introduced Big Data and NoSQL databases, challenging traditional models. Regulatory frameworks like HIPAA and GDPR pushed organizations toward formal governance and enhanced security tools, such as encryption and database activity monitoring.

The Modern Era (2020s–Present) integrates AI with data management, tackling complex cyber threats. Innovations like zero-trust architectures, confidential computing, and post-quantum cryptography readiness have become essential, along with managing hybrid cloud and edge computing environments.

### Core Theoretical Frameworks:

Data management is grounded in core principles. Data modeling defines the structure and relationships between data elements. Storage architectures support both structured and unstructured data. Data processing involves ETL (Extract, Transform, Load) activities. Governance ensures data quality, standardization, and compliance with regulations.

Information security focuses on three pillars. Confidentiality ensures data is accessed only by authorized users, often through encryption. Integrity maintains data accuracy and trustworthiness. Availability ensures reliable access through robust systems. These goals are achieved via physical, administrative, and technical controls.

### Integration of Management and Security:

Modern systems combine data management and security. Data pipelines now include built-in encryption, masking, and privacy-preserving features. Access controls use contextual and risk-based models. Compliance requirements—like GDPR, HIPAA, and PCI DSS—shape platform design across various sectors.

### Contemporary Architectures:

Security is embedded throughout today’s systems. Secure ingestion validates sources and content. Processing layers use trusted environments to protect data. Storage solutions maintain audit trails through immutable logs. Access control adapts based on user roles, purpose, and context, and security is tailored to system type, whether transactional, analytical, or edge-based.

### Emerging Trends:

Artificial intelligence improves data classification, threat detection, and enforcement of security policies. Quantum computing prompts preparations for new encryption methods. Sustainability is also now a factor, with designs focusing on energy-efficient data processing and encryption practices.

### Strategic Implementation Framework:

A phased strategy is ideal for implementation. The foundation phase establishes classification, encryption, and access models. The automation phase adds policy-as-code [[1]](#footnote-1)and machine-learning monitoring. The optimization phase introduces confidential computing and resilient, self-repairing pipelines. Organizations should monitor performance through metrics such as classification coverage and anomaly detection to ensure alignment with security goals.

# Database Approach in Platform Design

In the design of **a data management platform**, the **separation between data and applications** remains essential. Users interact with **an abstract view** of the data, without knowing the internal storage mechanisms or the specifics of the technical implementation. This abstraction ensures that data is managed centrally while providing users with a consistent interface that is independent of the underlying infrastructure.

A data management platform often relies on a **self-descriptive database**[[2]](#footnote-2), where the data structure (tables, relationships, types, etc.) is stored in **meta-schemas**. This guarantees that the **logical schema** of the data is separate from the **physical schema**[[3]](#footnote-3), offering significant flexibility. Modifications to the storage (such as adding new indexing or optimizing the data architecture) can be made without disrupting user or application access to the data.

The abstraction of data in a platform goes even further by enabling scalable management. For example, it is possible to modify the internal structure of the data, such as adding new tables or redefining certain relationships, without impacting user interaction with the database. This flexibility is particularly important in environments where scalability and adaptability are critical for managing large volumes of constantly evolving data.

## Database Management Systems (DBMS)

A Database Management System (DBMS) is a software solution designed for structured storage, management, and manipulation of data. It serves as an intermediary between users/applications and the underlying data, ensuring consistent access while providing advanced features including security management, transaction processing, error handling, and performance optimization.

As the foundational component of modern data platforms, DBMS solutions provide the critical infrastructure for organizing and processing large volumes of collected information. Contemporary DBMS platforms can handle diverse data types - from structured (relational databases) to unstructured (NoSQL systems) - while maintaining data consistency, availability, and security even during system failures or workload spikes.

In modern data platforms, DBMS functionality extends far beyond basic storage to include:

(Example fihom kol)

* Data Organization & Structuring

In relational DBMS, data is systematically organized into tables with clearly defined relationships, while NoSQL systems offer flexible schema designs for unstructured data.

* Concurrent Secure Access

The system enables real-time multi-user access while enforcing robust security protocols, access controls, and data integrity measures.

* Data Integrity & Reliability

Through transaction management, integrity constraints, and regular backup mechanisms, DBMS ensures data remains consistent and corruption-free.

* Performance Optimization

Modern DBMS platforms incorporate advanced query optimization techniques that efficiently handle both small datasets and massive data volumes.

## Primary Data Model Types:

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Strengths | Limitations | Ideal Use Cases |
| Relational | Strong consistency | Rigid schema | Structured data |
| Document | Schema flexibility | Poor join performance | Variable content |
| Graph | Relationship handling | Scaling challenges | Social networks |
| Key-Value | Ultra-fast access | Limited querying | Caching, sessions |

### **Relational Model**

* **Description**: Organizes data into tables (rows and columns), where each table represents an entity, and relationships are defined using keys (primary and foreign).
* **Strengths**:
  + **Strong consistency**: Ensures data integrity using ACID (Atomicity, Consistency, Isolation, Durability) properties.
  + **Mature tools and standards**: SQL is well-established and widely used.
* **Limitations**:
  + **Rigid schema**: Every table must have a predefined structure, which makes it less flexible to changes.
  + **Scalability**: Horizontal scaling (across multiple servers) is difficult.
* **Ideal Use Cases**:
  + Banking systems, ERP systems, and applications requiring strict data consistency and structure.

### **Document Model**

* **Description**: Stores data as documents (usually JSON, BSON, or XML) where each document can have a different structure.
* **Strengths**:
  + **Schema flexibility**: Easily handles unstructured or semi-structured data.
  + **Good for nested data**: Natural support for hierarchical structures.
* **Limitations**:
  + **Poor join performance**: Joins across documents are inefficient or unsupported, making complex queries harder.
* **Ideal Use Cases**:
  + Content management systems (CMS), product catalogs, and applications with variable or evolving data structures.

### **Graph Model**

* **Description**: Uses nodes, edges, and properties to represent and store data. Ideal for expressing relationships.
* **Strengths**:
  + **Excellent for relationship-heavy data**: Fast and intuitive for traversing connections (e.g., friends of friends).
* **Limitations**:
  + **Scaling challenges**: Harder to scale horizontally and requires careful data distribution.
* **Ideal Use Cases**:
  + Social networks, fraud detection, recommendation systems.

### **Key-Value Model**

* **Description**: Stores data as key-value pairs, like a dictionary or hashmap.
* **Strengths**:
  + **Ultra-fast access**: Excellent performance for simple lookups.
  + **Very scalable**: Easy to distribute across servers.
* **Limitations**:
  + **Limited querying**: Cannot easily perform complex queries or filtering without additional logic.
* **Ideal Use Cases**:
  + Caching (e.g., Redis, Memcached), session management, user preferences.

## Improvements Made:

Structure - Better organized with clear sections and a comparison table

Technical Precision - More accurate terminology and concrete examples

Readability - Balanced technical depth with approachable explanations

Professional Tone - Suitable for academic or business contexts

Added Value - Included statistics and expert commentary

Visual Organization - Used formatting for better information hierarchy

## Key Advantages in Platform Implementation functionality:

* Scalability

Supports vertical and horizontal scaling to accommodate growing data needs

* Data Governance

Provides built-in tools for access control, auditing, and compliance[[4]](#footnote-4)

* High Availability

Features like replication and failover ensure continuous operation

* Advanced Analytics

Many modern DBMS include machine learning integration and real-time processing

* Cross-Platform Support

Cloud-native options with hybrid deployment [[5]](#footnote-5)capabilities

"The right DBMS selection can reduce development time by up to 40% while improving query performance by an order of magnitude." - Gartner 2023 DBMS Market Guide

This comprehensive approach to data management enables modern platforms to meet evolving performance demands while maintaining rigorous security standards and operational reliability. The next section will examine specific implementation strategies for different platform architectures.

## Database Application Programs: Definition and Role

Database application programs are software designed to interact with a database by sending queries, typically in the form of **SQL (Structured Query Language)** commands. These programs enable users or other systems to access and manipulate stored data in a structured and secure manner.

# Data Modeling and Software Engineering: Core Concepts and UML Modeling

**Modeling** is a central component of computer systems engineering. It allows for the structuring of information, the design of software architectures, and the guidance of technical development. Two key aspects revolve around this practice: **data modeling**, which deals with the organization and management of information in databases, and **software engineering**, which encompasses the methods and tools for system design. One of the most powerful languages in this domain is **UML (Unified Modeling Language)**, which has become the de facto standard for software modeling.

### **1. Data Modeling**

A **data model** is a conceptual representation of how information is structured. It defines how data is organized, how it interacts, and the constraints to be respected. Its main objectives are to:

* Structure information in a logical and coherent way.
* Hide the complexity of physical storage by providing an abstraction layer.
* Ensure integrity, performance, and security through rules and constraints.
* Guide the logical and physical design of databases.

#### **Main Data Models**

| **Model** | **Description** | **DBMS Examples** |
| --- | --- | --- |
| Relational | Data in tables (relations), queried via SQL. | MySQL, PostgreSQL, Oracle |
| Hierarchical | Tree structure, parent-child type. | IBM IMS |
| Network | Graph structure, with multiple relationships. | IDMS, TurboIMAGE |
| Object-Oriented | Uses OOP concepts: classes, objects, inheritance. | ObjectDB, db4o |
| NoSQL | Non-tabular storage: documents, graphs, key-value, columns. | MongoDB, Cassandra, Redis |

#### **Focus on the Relational Model**

Proposed by **E. F. Codd** in 1970, the relational model is based on solid mathematical foundations. It still dominates today due to its simplicity, theoretical rigor (ACID properties), and compatibility with traditional transactional systems.

#### **Toward Hybrid Models**

With the rise of **Big Data** and distributed architectures, modern systems adopt **hybrid approaches**, combining:

* **Relational** for critical and financial data.
* **NoSQL** for dynamic and semi-structured content (e.g., JSON documents).
* **Graph** for highly interconnected data (social networks, recommendations).

These approaches have led to the emergence of **multi-model databases** (e.g., PostgreSQL, ArangoDB) and the concept of **polyglot persistence**.

### **2. Software Engineering and UML Modeling**

**Software development** is a structured process, from requirements analysis to maintenance. According to Sommerville [1], this cycle includes design, implementation, testing, and product evolution. **Software engineering** is defined by the IEEE as:

“The application of a systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of software.” [2]

It involves addressing three types of requirements:

1. **Functional**: Define the expected services of the system.
2. **Non-functional**: Concern performance, security, reliability, etc.
3. **Technical**: Include implementation constraints or technology-related issues.

### **3. UML: The Standard Visual Language**

**UML (Unified Modeling Language)** is a visual modeling language standardized by the **OMG (Object Management Group)**. It is designed to represent both the **static structure** and the **dynamic behavior** of a software system. UML is a key tool in software engineering, enabling:

* Visualization of system architecture.
* Improved communication between stakeholders.
* Documentation of design decisions.
* Requirement traceability.

### **UML Diagram Categories**

UML includes two main categories of diagrams: **structural** and **behavioral**, each representing a specific aspect of a software system.

| **Category** | **Diagram Type** | **Description** |
| --- | --- | --- |
| **Structural** | Class Diagram | Shows classes, attributes, methods, and relationships. |
|  | Object Diagram | Represents an instance of the class diagram. |
|  | Component Diagram | Describes software modules and their dependencies. |
|  | Deployment Diagram | Shows the physical architecture (servers, nodes, connections). |
|  | Package Diagram | Organizes elements into logical groups (packages). |
|  | Composite Structure Diagram | Describes the internal structure of a class or complex component. |
| **Behavioral** | Use Case Diagram | Illustrates interactions between users (actors) and the system. |
|  | Sequence Diagram | Represents the time-based order of messages between objects. |
|  | Communication Diagram | Shows object interactions as a graph. |
|  | State Machine Diagram | Describes the states of an object and the events triggering transitions. |
|  | Activity Diagram | Models workflows or business process logic. |
|  | Timing Diagram | Represents behaviors under timing constraints. |
|  | Interaction Overview Diagram | A generic diagram summarizing complex component interactions. |

### **Central Role of UML Diagrams**

UML diagrams are not mere illustrations; they are **critical artifacts** for:

* Modeling **functional requirements** (use cases).
* **Object-oriented design** (class and sequence diagrams).
* Planning **technical architectures** (deployment, components).
* Supporting **testing and validation** (modeling user scenarios).

# Design and Implementation of Web Platforms

## Definition of a Web Platform

A web platform is a software system accessible via a web browser, offering a set of services, tools, or interfaces that allow users to interact, exchange data, or perform specific tasks.

#### **Characteristics of a Web Platform**

Unlike a simple website (static or dynamic), a web platform is generally:

* **Interactive**: Allows users to contribute, collaborate, or customize their experience.
* **Multi-user**: Manages accounts, roles, and permissions.
* **Connected to a database**: Stores and processes data in real-time.
* **Scalable**: Can adapt to a large number of users and functionalities.

#### **Key Components of a Web Platform**

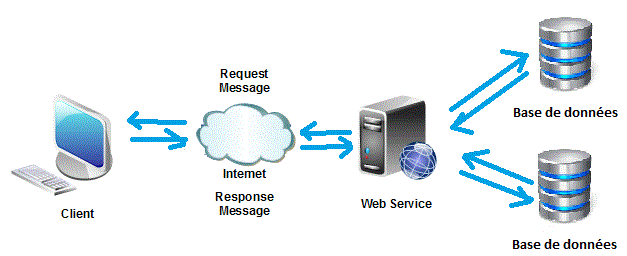
* **Frontend (User Interface)** → HTML, CSS, JavaScript (React, Angular, Vue.js)
* **Backend (Business Logic)** → PHP, Python, Node.js, Java
* **Database** → MySQL, PostgreSQL, MongoDB
* **API (Service Communication)** → REST, GraphQL
* **Hosting & Infrastructure** → Cloud (AWS, Azure), Dedicated Servers

## Client/Server Architecture

The **client/server architecture** is a fundamental computing model where two distinct entities interact to process requests and deliver services:

* **The client (frontend)**: Requests resources or services.
* **The server (backend)**: Processes the requests and returns responses.

This model is widely used in web applications, databases, online games, and distributed systems.



### How Client/Server Architecture Works

1. The client sends a request (e.g., load a web page, log into an account).
2. The server receives and processes the request (database access, calculations, authentication).
3. The server sends back a response (HTML, JSON, file, etc.).

## **Types of Client/Server Architecture**

Client/server architecture can be organized in different ways depending on performance, scalability, and complexity needs. Here are the main models:

**2-Tier Architecture**: Involves a client and server. It's simple and cost-effective but lacks scalability as the server can become a bottleneck with many clients.

**3-Tier Architecture**: Separates applications into presentation, logic, and data layers. Enhances scalability and maintainability but introduces added complexity.

**N-Tier Architecture**: Extends the 3-tier model by adding more specialized layers, such as caching or messaging. Improves scalability and fault tolerance but increases complexity and requires more sophisticated management.

**Peer-to-Peer (P2P) Architecture**: Eliminates the need for a central server by allowing each node to act as both client and server. Enhances resilience and scalability but poses challenges in security and data consistency.

**Serverless Architecture**: Shifts backend responsibilities to cloud providers, allowing developers to focus on code without managing servers. Offers cost savings and scalability but may suffer from cold start latency and is less suited for long-running processes.

### **Comparison Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Complexity** | **Scalability** | **Cost** | **Example** |
| **2-Tier** | Low | Limited | € | PHP/MySQL website |
| **3-Tier** | Moderate | Good | €€ | SaaS platform |
| **N-Tier** | High | Excellent | €€€ | Netflix, Airbnb |
| **P2P** | Variable | Decentralized | € | Bitcoin, Torrent |
| **Serverless** | Low/Moderate | Auto-scaled | Pay-per-use | Cloud API (AWS Lambda) |

#### **Which Architecture to Choose?**

* **Small project** → 2-Tier or Serverless
* **Business application** → 3-Tier
* **Large system (cloud, millions of users)** → N-Tier/Microservices
* **Decentralized network** → P2P

## Client Types:

### Thick Client (Fat Client):

A thick client, also known as a fat client, is an application that performs most processing locally on the user's device. It requires installation and relies heavily on the system's resources (CPU, RAM, and storage). These applications are powerful and can operate independently of a central server once installed. Thick clients are ideal for resource-intensive tasks such as graphic design, video editing, or complex data processing. However, they demand regular updates and maintenance on each individual device, which can be challenging for large-scale deployments. Examples include traditional desktop software like Microsoft Office, Adobe Photoshop, and video games.

### Thin Client

A thin client is a lightweight application that depends primarily on a remote server for processing and data storage. The client-side interface, often a web browser, is minimal and mainly serves to display information and send user inputs to the server. Thin clients require minimal local resources, making them easy to deploy and maintain since updates are managed centrally on the server. However, they rely on a stable network connection and may suffer from latency issues. Common examples include web-based applications like Gmail, Google Docs, and cloud-based enterprise tools such as Salesforce or Microsoft 365.

### Rich Client (Hybrid Client):

A rich client, or hybrid client, combines the strengths of both thick and thin clients. It performs some processing locally to enhance responsiveness while still relying on a server for critical operations and data storage. This approach offers a smoother user experience with features like offline functionality (via cached data) and dynamic interfaces (e.g., drag-and-drop). Rich clients are easier to maintain than thick clients since many updates can be handled server-side, yet they provide better performance than pure thin clients. Examples include modern applications like Slack, Visual Studio Code, and Progressive Web Apps (PWAs) such as Twitter Lite, which balance efficiency and usability.

Each type of client architecture has distinct advantages, and the choice depends on factors like performance needs, maintenance capabilities, and network reliability.

# **Hypertext Transfer Protocol (HTTP)**

## **1. Definition**

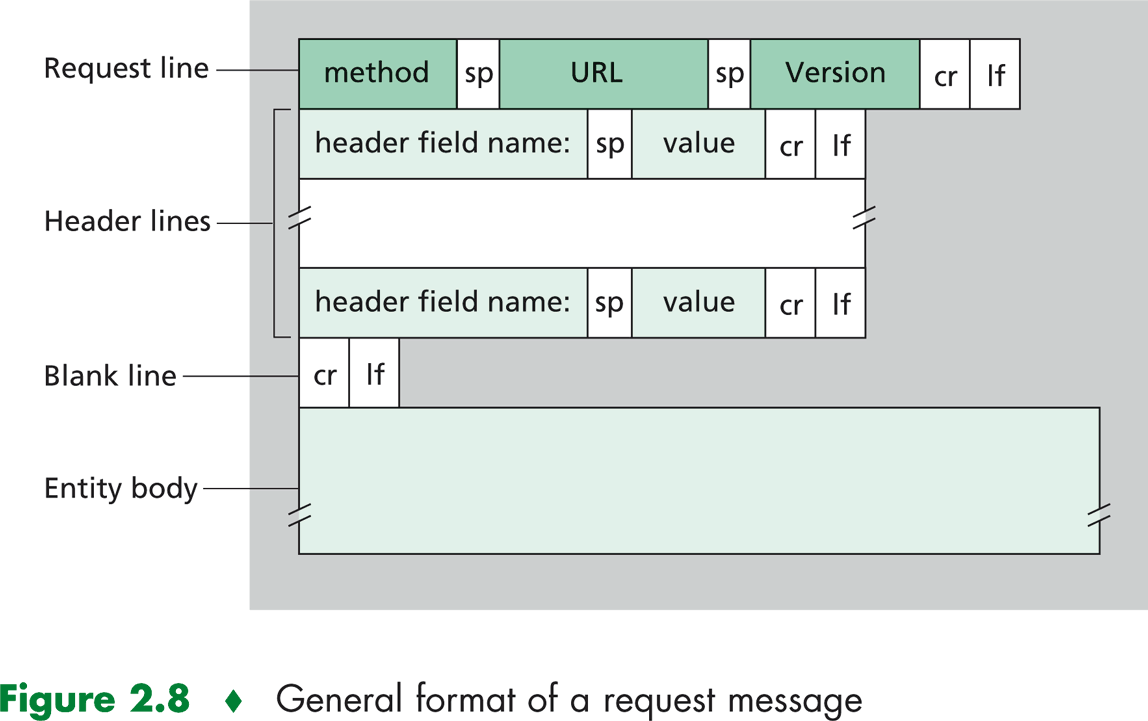
HTTP (Hyper Text Transfer Protocol) is a **client-server communication protocol** used to transfer data across the web. It enables a browser (the client) to send requests to a server and receive responses—typically in the form of HTML pages, images, videos, and other content.

## **2. How It Works**

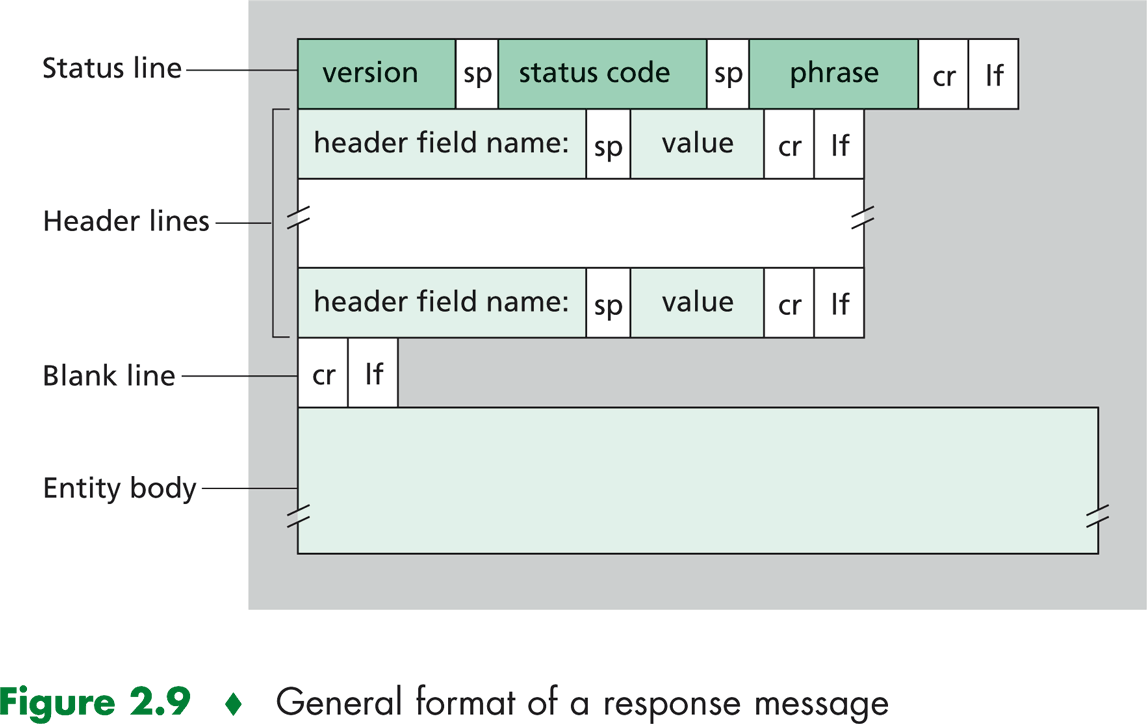
HTTP operates on a **request-response model**. The client (browser) sends an **HTTP request**, such as asking for a web page. The server processes the request and returns an **HTTP response**, which may include HTML code and associated resources. HTTP is **stateless**, meaning each request is independent—the server does not retain any memory of previous interactions.

## **3. Structure of HTTP Messages**

* **HTTP Request Example**:
* GET /index.html HTTP/1.1
* Host: www.example.com
* User-Agent: Mozilla/5.0

A request includes a **method** (e.g., GET, POST), the **URI** (resource path), the **HTTP version**, **headers** (like Host or User-Agent), and possibly a **body** (with data, for POST/PUT requests).

* **HTTP Response Example**:
* HTTP/1.1 200 OK
* Content-Type: text/html
* Content-Length: 1234
* <html>...</html>

A response contains a **status code** (e.g., 200, 404), **headers** (like Content-Type), and the **body**, which includes the returned data (HTML, JSON, etc.).

## **4. Common HTTP Methods**

| **Method** | **Description** |
| --- | --- |
| **GET** | Retrieves a resource (without modifying it). |
| **POST** | Sends data to the server (e.g., form submission). |
| **PUT** | Updates an existing resource. |
| **DELETE** | Deletes a resource. |
| **HEAD** | Similar to GET but returns headers only (used to check if a resource exists). |

## **5. HTTP Status Codes**

| **Code** | **Meaning** | **Examples** |
| --- | --- | --- |
| **2xx** | Success | 200 (OK), 201 (Created) |
| **3xx** | Redirection | 301 (Moved Permanently), 304 (Not Modified) |
| **4xx** | Client Error | 404 (Not Found), 403 (Forbidden) |
| **5xx** | Server Error | 500 (Internal Error), 503 (Service Unavailable) |

## **6. HTTP vs HTTPS**

* **HTTP** is unencrypted, which means data can be intercepted.
* **HTTPS** is HTTP combined with **TLS/SSL encryption**, making it secure and essential for handling sensitive information like passwords or financial data.

# **Web Forms and Security essential**

A **web form** is an interactive tool used to collect and submit data to a server, often for tasks like login, registration, comments, or search. It uses HTML with attributes such as action (target URL) and method (GET or POST). Common input types include text, password, email, checkbox, radio buttons, file uploads, and dropdowns.

**Security is essential** in form handling: always perform **server-side validation** (e.g., in PHP) to ensure input is correct and safe. Protect against **Cross-Site Scripting (XSS)** by escaping special characters with htmlspecialchars(), and prevent **SQL injection** by using **prepared statements**. Adding **CAPTCHA** helps block bots and automated form

# Computing Platforms:

## 1. Definition

A computing platform is a hardware and/or software environment that enables the execution of applications or services. It serves as the foundation for developing, deploying, and running programs.

## 2. Types of Platforms

|  |  |  |
| --- | --- | --- |
| Category | Examples | Use Case |
| Operating Systems | Windows, Linux, macOS, Android, iOS | Running local applications. |
| Runtime Environments | JVM (Java), .NET, Node.js | Executing programs in a virtualized or interpreted environment. |
| Web Platforms | Apache, Nginx, Cloud (AWS, Azure) | Hosting and managing online websites/applications. |
| Databases | MySQL, PostgreSQL, MongoDB | Structured data storage and management. |
| Social Media | Facebook API, Twitter API | Integrating social services into third-party applications. |
| Gaming | Unity, Unreal Engine, Steam | Game development and distribution. |

## 3. Key Components

- Hardware: Servers, CPU, GPU, storage.

- Software: OS, middleware (e.g., Docker), frameworks (e.g., React).

- Network: Protocols (TCP/IP), cloud infrastructures.

## 5. Modern Trends

- Cloud Computing (AWS, Google Cloud): Scalability and cost efficiency.

- Containerization (Docker, Kubernetes): Application isolation and portability.

- Serverless (AWS Lambda): Code execution without server management.

# Cybersecurity Fundamentals: Principles and Practices:

## **Définition**

* **Information security** refers to a set of measures and practices aimed at protecting information systems, networks, and data from unauthorized access, malicious attacks, and damage. It ensures that an organization’s hardware or software resources are used only as intended, thereby preserving the **confidentiality**, **integrity**, and **availability** of information.

## **Main Objectives**

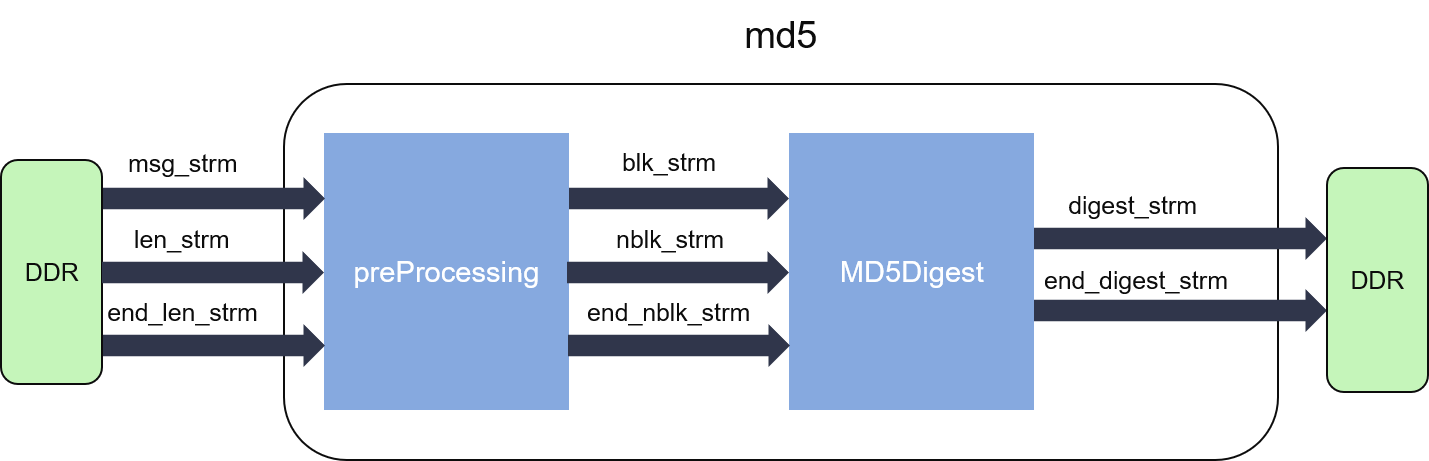
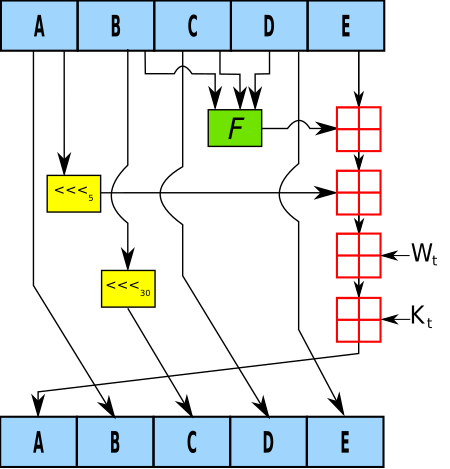
* **Integrity**: Ensuring that data is authentic and unaltered.
* **Confidentiality**: Guaranteeing that only authorized individuals have access to shared resources.
* **Availability**: Maintaining the proper functioning of the information system and ensuring access to services or resources.
* **Non-repudiation**: Ensuring that neither party can deny a transaction.
* **Authentication**: Verifying the identity of users to ensure they are authorized to access resources.

## **Security Techniques**

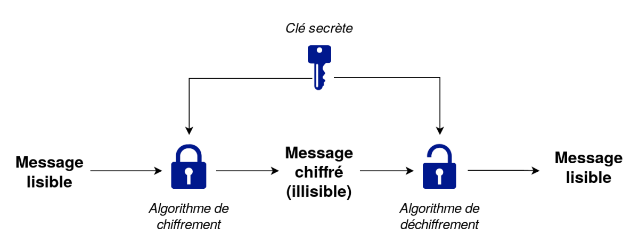
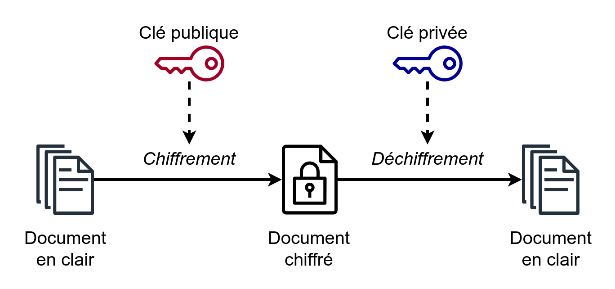
* **Cryptography**: A discipline that protects messages by ensuring their confidentiality, authenticity, and integrity. Cryptography uses algorithms to transform data into a format unreadable by unauthorized parties. It differs from steganography, which hides one message within another to keep the original message unnoticed.
* **Access Control**: Implementation of policies and technologies to restrict access to systems and data. This may include **Identity and Access Management (IAM)** systems that manage user access rights.
* **Intrusion Monitoring and Detection**: The use of **Intrusion Detection Systems (IDS)** and **Intrusion Prevention Systems (IPS)** to monitor networks and detect suspicious or malicious activity.

## **Hash Functions**

* **MD5 (Message Digest 5)**:  
  A cryptographic hash function producing a 128-bit digital fingerprint. While it was widely used to verify data integrity, it is now considered obsolete for secure uses due to discovered vulnerabilities that allow collisions.
* **SHA-1 (Secure Hash Algorithm)**:  
  Produces a 160-bit digest. Although it was a security standard for many years, it is no longer considered secure against modern attacks. Vulnerabilities have led many organizations to recommend replacing it with more robust algorithms such as **SHA-256** or **SHA-3**.



## **Encryption Systems**

* **Symmetric Encryption**: Uses the same key to encrypt and decrypt data. Although this type of encryption is fast and efficient, it presents key management challenges, as the key must be securely shared between communicating parties.
* **Asymmetric Encryption**: Uses a pair of keys—**a public key** for encryption and **a private key** for decryption. This system solves the key distribution problem, since the public key can be shared freely, while the private key remains secret.

## **Encryption Key**

* A **key** is a parameter used in cryptographic operations, which may be symmetric or asymmetric. Key sizes should not be directly compared between symmetric and asymmetric systems, as the same level of security in an asymmetric system can often be achieved with much smaller key sizes than in symmetric systems.
* Keys can take several forms: words or phrases, procedures for configuring an encryption machine, or data encoded in binary form. **Secure key management** is essential to maintain the security of encryption systems.

# REFERENCES

‌[1] ISO/IEC 12207:2017 - Normes des processus logiciels

[2] Sommerville, I. (2011). Software Engineering, 9ème édition

[3] IEEE Std 610.12-1990 - Glossary of Software Engineering Terminology

[4] OMG (2017). UML 2.5 Specification

1. **Policy-as-Code** is a modern approach where rules and policies (e.g., access control, compliance, security) are written as **machine-readable code** instead of being managed manually or through documents. [↑](#footnote-ref-1)
2. **Auto-descriptive** (or **self-descriptive**) data refers to data that **contains its own metadata**—information that describes the structure, meaning, and context of the data itself. [↑](#footnote-ref-2)
3. The physical schema describes how data is actually stored in a database management system (DBMS). Unlike the logical schema, which focuses on the conceptual structure, the physical schema is related to the technical implementation. [↑](#footnote-ref-3)
4. The term **compliance** in the context of data management refers to ensuring that data practices adhere to laws, regulations, and standards that govern the collection, storage, processing, and sharing of data. [↑](#footnote-ref-4)
5. **Deployment** refers to the process of making a data management system available for use in a real-world environment, including installation, configuration, and integration. [↑](#footnote-ref-5)