LECTURER: Anna Androvitsanea

DATA ENGINEERING

Database Engineering — Unit 1

Dr.-Ing Anna Androvitsanea

IU Internationale Hochschule GmbH

January 29, 2024

DATA ENGINEERING
TOPIC OUTLINE

Introductory Round

- Who are you?
- Anna
- iu / Flightright GmbH
- Data Scientist and ML Engineer
- Have worked as a Civil Engineer and Archaeologist
- Data Scientist enthusiast

Role of Data Engineers and Data Scientists

- Exploring the distinct roles and responsibilities of data engineers and data scientists.
- Examining the interplay and collaboration between these two roles in a real-world project.
- Case Study: A successful collaboration between data engineering and data science teams in a retail analytics project.

Case Study Overview: Retail Analytics Project

- Introduction to the retail analytics project.
- Objectives and challenges of the project.
- Overview of the teams involved: Data Engineering and Data Science.

Role of the Data Engineering Team

- Responsibilities and tasks of the data engineering team in the project.
- Techniques and technologies used for data collection, storage, and preprocessing.
- How the team ensured data quality and integrity.

Role of the Data Science Team

- The data science team's approach to data analysis and modeling.
- Specific data science methods and tools used.
- Insights and predictions derived from the data.

Collaboration Between Teams

- How the teams worked together: communication and workflow strategies.
- Integration of data engineering outputs with data science models.
- Addressing challenges and adapting strategies in the collaborative process.

Outcomes and Achievements

- Key results and successes of the project.
- Impact of the collaboration on project outcomes.
- Lessons learned and best practices identified for future projects.

Unit 1

FOUNDATIONS OF DATA SYSTEMS

Study Goals

- Explain properties of well-designed data-intensive systems.
- Learn how to quantify the reliability of a system.
- Understand how to compare different approaches to system scalability.
- Discuss aspects of maintainability in the development of data-intensive systems.

EXPLAIN SIMPLY

1. List **three aspects** that enable modern data-intensive data systems to process large amounts of data while running stably.
2. Explain ways to **adapt** these systems to higher or lower **performance requirement**s.
3. Discuss how these systems can be designed for straightforward operation in terms of **maintenance and evolution**.

RECAP OF THE UNIT'S KEY TOPICS

Reliability	Scalability	Maintainability
• Reliability Metrics • Hardware Fault • Software Error • Human Error	• Load and Throughput • Service Level Agreements (SLAs) • Scale Up – Scale Out	• Operability • Simplicity • Evolvability

Source of the text: Hartz, Walker & Mahar, 1997.

Properties of Well-Designed Data-Intensive Systems: Reliability

- Reliability: Ability to function correctly even in the face of adversity.
- Includes handling user errors, system faults, and environmental challenges.

Scalability

- Scalability: Capacity to handle growth and increased load effectively.
- Involves scaling up/out and managing load distribution efficiently.

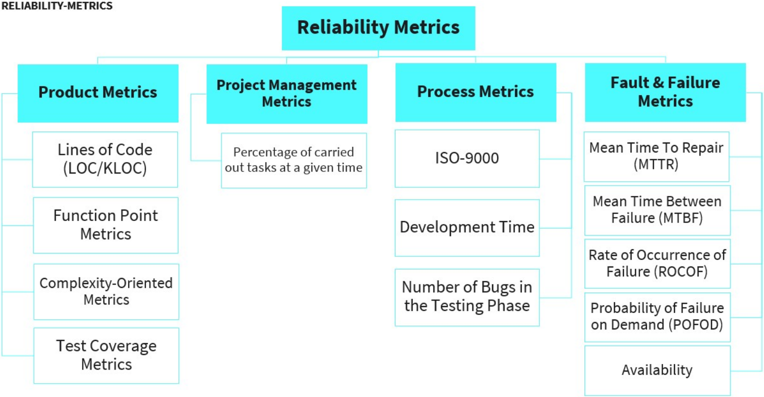
Maintainability

- Maintainability: Ease of maintaining system operations and updates.
- Focuses on simplicity, good documentation, and automation of operational tasks.

Properties of Well-Designed Data-Intensive Systems

Reliability

RELIABILITY-METRICS



Source of the text: Wilkins, 2002.
Source of the image: Müller-Kett (2022) based on Hartz et al., 1997.

Reliability Metric: Lines of Code (LOC/KLOC)

- **Explanation:** LOC (Lines of Code) and KLOC (Thousand Lines of Code) measure software size or complexity.
- **Example:** Using LOC to estimate effort or complexity of software projects. Larger projects typically have more LOC.

Reliability Metric: Function Point Metrics

- **Explanation:** Measures software functionality from the user's perspective.
- **Example:** Used in budgeting and planning, assessing the functionality delivered to users.

Project Management Metric: Development Time

- **Explanation:** Tracks the total time taken from project initiation to completion.
- **Example:** Used to assess project efficiency, schedule adherence, and predict future project timelines.

Process Metric: Number of Bugs in Testing Phase

- **Explanation:** Counts the number of defects found during the testing phase of software development.
- **Example:** Indicator of software quality before release; higher numbers may indicate issues in earlier development stages.

Fault Failure Metric: Mean Time To Repair (MTTR)

- **Explanation:** Measures the average time taken to repair a fault or failure in the system.
- **Example:** Used in maintenance and reliability analysis; shorter MTTR implies better system maintainability.

Fault Failure Metric: Mean Time Between Failures (MTBF)

- **Explanation:** Indicates the average time between failures in a system.
- **Example:** Used to gauge the reliability of a system; longer MTBF is desirable.

Fault Failure Metric: Rate of Occurrence of Failure (ROCOF)

- **Explanation:** Measures the frequency of failures in a system.
- **Example:** Useful for identifying trends in system reliability over time.

Fault Failure Metric: Probability of Failure on Demand (POFOD)

- **Explanation:** Assesses the likelihood of a system failing when required.
- **Example:** Critical for systems where reliability on demand is essential, like safety-critical systems.

Fault Failure Metric: Availability

- **Explanation:** Measures the proportion of time a system is operational and available for use.
- **Example:** High availability is crucial for systems requiring constant uptime, like web servers.

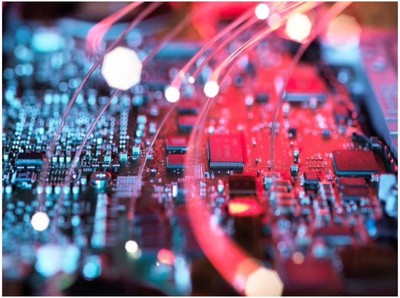
HARDWARE FAULT

Physical damage in one machine's components

- Redundancies (i.e., RAIDs)
- Redundancy across multiple machines

Power supply blackouts / Network disruptions

- Redundancies across multiple data centers/availability zones



Source of the text: Détienne, 2002.
Source of the image : Microsoft Archive.

Hardware Faults

- Physical damage in one machine's components.
- Strategies to mitigate:
  - Implementing redundancies, like RAID systems.
  - Redundancy across multiple machines.

Understanding RAID Systems

- RAID (Redundant Array of Independent Disks) is a data storage technology.
- Combines multiple physical disk drive components into one or more logical units.
- Key Purposes:
  - Data redundancy: Protects data by duplicating it across multiple disks.
  - Improved performance: Can increase read and write speeds by distributing operations across several disks.
- Common RAID Levels:
  - RAID 0: Striped set without parity or mirroring.
  - RAID 1: Mirroring without parity or striping.
  - RAID 5: Block-level striping with distributed parity.

Redundancies

- Essential for mitigating hardware and network failures.
- Types of Redundancies:
  - In-device redundancies (like RAIDs).
  - Across multiple devices or systems.
  - Redundancies across data centers or availability zones.

Network Disruptions

- Causes and consequences of network disruptions.
- Mitigation strategies:
  - Redundancies across multiple data centers.
  - Diverse network routes to prevent single points of failure.

SOFTWARE ERROR

- Memory leakage
- Service stops working
- Cascading failure

- Software is a mental product and humans will introduce error to this

Source of the image: Naylor & Joyner, 2014.
Source of the image : Microsoft Archive.

MENSCHLICHER ERROR

- Humans might introduce considerable potential for error into systems
- 32% of security incidents are caused by employees

- Provide efficient recovery tools
- Provide minimum opportunity for human error
- Intensive testing

Source of the text: BakerHostetler, 2017.
Source of the image : Microsoft Archive.

- **System Load**
  - Read-to-Write ratio
  - Number of online users
  - Requests per time
- **System Performance Load**
  - Running times for read/write operations
  - Throughput
  - I/O-Rate: Input-Ouput Operations per Second (IOPS)
  - Response Time

Properties of Well-Designed Data-Intensive Systems
Scalability

System Load

- System Load refers to the volume of work handled by a system.
- Key Metrics:
  - Read-to-Write ratio: Balance of reading vs. writing operations.
  - Number of online users: Concurrent users interacting with the system.
  - Requests per time: Frequency of operations requested from the system.

System Performance Load

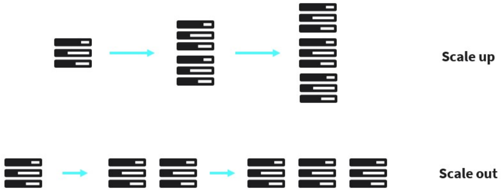
- Focuses on the efficiency of processing operations.
- Measures:
  - Running times for read/write operations.
  - How quickly the system can process tasks.

Throughput

- Throughput: Quantity of work a system can handle in a given time.
- Key Indicators:
    - I/O-Rate (Input-Output Operations per Second, IOPS).
    - Response Time: The time it takes for the system to respond to a request.

SCALE UP – SCALE OUT

- Optimize the **software-side**
- Scale the **physical ressources**



Source of the image: Alvarid (2022).

Scale Up: Optimizing Software and Resources

- **Definition:** Scale Up (Vertical Scaling) involves increasing the power of existing hardware or software.
- **Optimize the Software-Side:**
  - Enhancing the performance of the application to better utilize existing resources.
  - Examples: optimizing algorithms, refining code for efficiency.
- **Scale the Physical Resources:**
  - Upgrading existing hardware capabilities: more RAM, faster CPU, larger storage.
  - Impact: Improved performance without changing the system architecture.

Scale Out: Expanding Capacity Horizontally

- **Definition:** Scale Out (Horizontal Scaling) involves adding more nodes to a system to distribute the load.
- **Optimize the Software-Side:**
  - Designing software to efficiently run on multiple servers or nodes.
  - Examples: Implementing load balancing, ensuring data consistency across nodes.
- **Scale the Physical Resources:**
  - Adding more machines or instances to the existing pool.
  - Impact: Enhances capacity and reliability, allows for handling increased load without a single point of failure.

Properties of Well-Designed Data-Intensive Systems

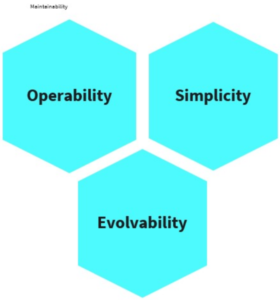
Maintainability

MAINTAINABILITY

Making it easy to install, use and modify the system

Minimizing the discomfort of legacy maintenance

- Corrective maintenance
- Adaptive maintenance
- Perfective maintenance
- Preventive maintenance



Source of the image: Müller-Kett (2022).

OPERABILITY

Operators are responsible for

- Monitoring the system's health
- Installing security patches
- Predicting and solving potential problems
- Ensuring a stable production environment
- Preserving the knowledge about the system

Source of the text: Hamilton, 2007, p. 231-242.
Source of the image: Müller-Kett (2022).

OPERABILITY

Making it easy for operators to use the system

- Providing transparent monitoring possibilities
- Supporting automation and integration with standard tools
- Providing concise documentation
- Ensuring service/machine independence
- Providing default setups and self-healing features

Source of the text: Hamilton, 2007, p. 231-242.
Source of the image: Müller-Kett (2022).

SIMPLICITY

Making the code as easy as possible and only as complex as it is necessary

Reducing…

- accidental complexity
- long code
- module interactions & dependencies
- inconsistent naming

- increasing the level of abstraction

Source of the image: Müller-Kett (2022).

Simplifying Code

- Aim: Make code as easy as possible, only complex as necessary.
- Benefits: Easier to understand, maintain, and less prone to errors.

Reducing Accidental Complexity

- Accidental Complexity: Unnecessary complexity not inherent to the problem.
- Strategies: Refactoring code, using design patterns, simplifying logic.

Streamlining Code Length and Modules

- Reducing Long Code: Break down into smaller, manageable functions.
- Managing Module Interactions: Minimize dependencies, use clear interfaces.

Consistent Naming and Abstraction

- Consistent Naming: Improves readability and maintainability.
- Increasing Level of Abstraction: Focus on the 'what' over 'how'.

EVOLVABILITY

Making changes to the system as easy as possible

- Similar to the term “agile”
- DataOps

Source of the image: Müller-Kett (2022).

Evolvability in Data Systems

- Objective: Making changes to the system as easy as possible.
- Similar to the agile methodology: Adaptability and responsiveness to change.
- Example: Iterative development, where systems are continuously improved.

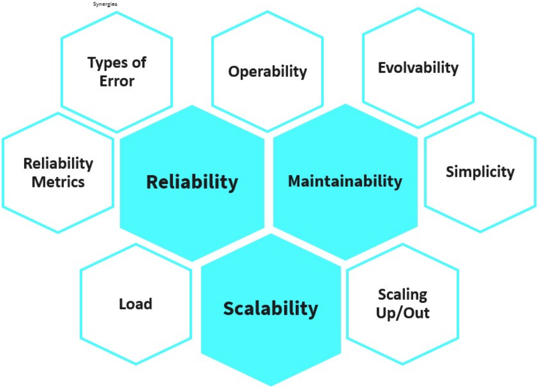
DataOps: Facilitating Evolvability

- DataOps: Agile and lean practices in data management.
- Example: Automating data pipelines to quickly adapt to new data sources.
- Benefits: Faster deployment, improved quality, and more efficient data use.

Example of a Data Pipeline

- **Objective**: Move and process data from source to destination.
- **Stages**:
  1. *Extraction*: Collect data from various sources (e.g., databases, online services).
  2. *Transformation*: Process data (e.g., cleaning, aggregating, formatting).
  3. *Loading*: Load transformed data into a destination (e.g., data warehouse).
- **Use Case**:
  - Extract sales data from online transactions.
  - Aggregate sales by region and calculate total revenue.
  - Load processed data into a data warehouse for analysis.

SYNERGIES



Source of the image: Müller-Kett (2022).

Study Goals

- Explain properties of well-designed data-intensive systems.
- Learn how to quantify the reliability of a system.
- Understand how to compare different approaches to system scalability.
- Discuss aspects of maintainability in the development of data-intensive systems.

Study Goals: Key Aspects of Modern Data-Intensive Systems

- **Distributed Computing**
  - Utilizes multiple machines for efficient data handling.
  - Improves resilience and scalability.
- **Advanced Data Storage Solutions**
  - Incorporates technologies like NoSQL databases.
  - Facilitates efficient data management and retrieval.
- **High-Performance Processing Frameworks**
  - Uses frameworks like Hadoop and Spark.
  - Optimized for rapid analysis and processing of big data.

Study Goals: Quantifying System Reliability

- Understanding Key Reliability Metrics:
  - Mean Time Between Failures (MTBF).
  - Mean Time To Repair (MTTR).
  - System Availability and Uptime.
- Applying Reliability Metrics:
  - Calculating reliability scores.
  - Analyzing failure rates and recovery processes.
- Case Study: Examining real-world systems' reliability reports.

Study Goals: Comparing Approaches to System Scalability

- Understanding Scalability:
  - Definition and importance in data-intensive systems.
- Scalability Approaches:
  - Vertical Scaling (Scaling Up): Increasing the power of existing hardware.
  - Horizontal Scaling (Scaling Out): Adding more nodes to a system.
- Evaluating Scalability Strategies:
  - Cost-effectiveness, performance implications, and maintenance considerations.
  - Real-world examples and case studies for context and understanding.

Study Goals: Maintainability in Data-Intensive Systems

- Importance of Maintainability:
  - Key to long-term success and adaptability of systems.
- Aspects of Maintainability:
  - Code Simplicity and Clarity: Easy to understand and modify.
  - Documentation Quality: Comprehensive and up-to-date.
  - Automated Testing: Ensures reliability and ease of changes.
  - Modular Design: Allows independent development and updates.
- Strategies for Improvement:
  - Regular code reviews, refactoring, and keeping up with technological advancements.

EXPLAIN SIMPLY

1. List **three aspects** that enable modern data-intensive data systems to process large amounts of data while running stably.
2. Explain ways to **adapt** these systems to higher or lower **performance requirement**s.
3. Discuss how these systems can be designed for straightforward operation in terms of **maintenance and evolution**.

EXPLAIN SIMPLY: Key Aspects for Processing Large Data Volumes

- **Distributed Computing**: Utilizing multiple machines for data processing.
- **Advanced Storage Systems**: Implementing scalable and efficient storage solutions like NoSQL.
- **High-Performance Processing**: Leveraging frameworks like Hadoop or Spark for fast data processing.

EXPLAIN SIMPLY: Adapting Systems to Performance Requirements

- **Scalability**: Vertical scaling (enhancing existing systems) and horizontal scaling (adding more systems).
- **Resource Optimization**: Tailoring resource allocation based on current load and performance targets.
- **Performance Tuning**: Adjusting configurations for optimal efficiency under varying workloads.

EXPLAIN SIMPLY: Designing for Maintenance and Evolution

- **Modular Architecture**: Facilitates updates and integration of new technologies.
- **Automation and Testing**: Ensures reliability and eases updates.
- **Documentation and Standardization**: Provides clarity and consistency for future development.

SESSION 1

TRANSFER TASK

A start-up that sells **sustainable products in smaller stores** has been very successful in recent years. As a result, more stores are to be opened worldwide. As a Data Engineer, you will be tasked with **designing the data system that stores and processes data** about the products offered and their suppliers.

As a team, develop **key points** to ensure that this system will run appropriately effectively and perform well. For each of these points, think about **specific measures to be implemented** in the system.

Please present your results.
The results will be discussed in plenary.

- **Reliability metrics** are implemented in the system
  - To ensure a fast implementation, the following metrics are implemented first (based on log file monitoring) and monitored in a dashboard, for example
  - Product metrics (lines of code, function point metrics, test coverage metrics)
  - Process metrics (number of bugs in the test phase)
  - Fault & Failure metrics (Availability, Rate of Occurrence of Failure (ROCOF), Mean Time To Repair (MTTR)
- To keep the system functional in case of **hardware failures**, it is implemented on multiple redundant computing units
  - Virtualization in containers
  - Distribution of the system over several data centers
  - Distribution of the system over several geographically separated "Availability Zones"

- The system is implemented in **collaboration** with an established **cloud provider**
  - This way, we ensure that the system is dynamically scalable at all times
- We use both **vertical and horizontal scaling** to adapt the system to the system load
- To maintain a defined **service level**, the system is constantly monitored for load and throughput based on log files
- Scale Up - Scale Out

We ensure the **operability** of the system by....

- making the system **easy to monitor** in the form of a dashboard and automated alert mechanisms
- **simplifying deployment** through a predefined CI/CD pipeline
- providing **clear documentation**
- **isolate** individual **services** from each other
- implementing predefined **default settings** that can be overridden if necessary

We keep the system **simple** by...

- regularly checking if the code can be **simplified**
- keeping the code as **short** as possible
- **avoiding interactions** between modules
- using **abstraction** (e.g. by self-contained functions and modules)
- set up rules for **consistent naming**

In addition, we design the system in such a way that it can be **easily modified or extended**

- for example, we implement all self-contained functionalities as **independent functions or modules**

LEARNING CONTROL QUESTIONS

1. Which task does not fall under the responsibility of data engineers?
   a) Data Analysis
   b) Removing corrupted data
   c) Acquiring data from various sources
   d) Optimizing databases for analysis

2. Which of the following choices is the term used to describe the probability of error-free software operation over a given period of time in a given environment?
   a) Reliability
   b) Maintainability
   c) Scalability
   d) Security

3. If some errors are found in the application and they need to be fixed, what kind of maintenance is used?
   a) adaptive maintenance
   b) perfecting maintenance
   c) preventive maintenance
   d) corrective maintenance

LIST OF SOURCES

Text:

BakerHostetler. (2017, May 9). *Webinar: Be compromise ready: Go back to the basics*. https://www.bakerlaw.com/events/webinar-be-compromise-ready-go-back-to-thebasics

Détienne, F. (2002). *Software design: Cognitive aspects*. Springer.

Hamilton, J. (2007). On designing and deploying internet-scale services. *Proceedings of the 21st Large Installation System Administration Conference (LISA '07)*, 231—242.

Hartz, M. A., Walker, E. L. & Mahar, D. (1997). *Introduction to software reliability: A state of the art review*. The Center.

Naylor, W., & Joyner, B. (2014). A discourse on software safety and software reliability.*2014 Reliability and Maintainability Symposium*. https://ieeexplore.ieee.org/document/6798493

Wilkins, D. J. (2002). The Bathtub Curve and Product Failure Behavior (Part 1 of 2). *Reliability HotWire*, *21*. Abgerufen am 15. März 2022, von https://www.weibull.com/hotwire/issue21/hottopics21.htm

Image:

Alavirad, 2020.
Müller-Kett (2022).
Müller-Kett (2022) based on Hartz et al., 1997.
Microsoft Archive.