Software Design and Architecture

İçindekiler Tablosu

Software Design and Architecture 1

1. **Introduction** 3
2. **Introduction to Software Architecture** 5
   - 2.1. **Importance of Software Architecture** 7
   - 2.2. **Why is Software Architecture Important?** 10
   - 2.3. **Role of Architects and Developers** 13
   - 2.4. **Software Architecture, Design and Implementation** 16
   - 2.5. **Software qualities (performance, scalability, maintainability, security)** 18
3. **Software Development Life Cycle (SDLC) and Architecture** 21
   - 3.1. **UML for Design in Software Engineering** 25
   - 3.2. **JIRA for Project Tracking in Software Engineering** 27
   - 3.3. **Waterfall, Agile, and DevOps Perspectives on Architecture** 30
   - 3.4. **Where Architecture Fits in SDLC (Software Development Life Cycle)** 33
   - 3.5. **Architectural Decision-Making Process** 36
4. **Software Design Principles** 41
   - 4.1. **Abstraction, Modularity, and Separation of Concerns (SoC)** 45
5. **Creating a Software Architecture** 48
   - 5.1. **Defining Requirements in Software Engineering** 50
   - 5.2. **Prioritizing Quality Features in Software Engineering** 53
   - 5.3. **Selecting the Architectural Style or Approach in Software Engineering** 56
   - 5.4. **Defining Architectural Components in Software Engineering** 60
   - 5.5. **Types Of Architecture** 63
     - 5.6.1. **Layered Architecture** 63
     - 5.6.2. **Microservices Architecture** 66
     - 5.6.3. **Client–Server Architecture** 69
     - 5.6.4. **Event-Driven Architecture (EDA)** 72
   - 5.7. **Selecting Technologies and Tools in Software Engineering** 74
     - 5.7.1. **Programming Languages** 76
     - 5.7.2. **Framework** 78
     - 5.7.3. **Libraries** 79

5.7.4. Databases......81
5.7.5. Development Environments......82
5.7.6. Platforms......83
5.8. Preparing Architectural Diagrams in Software Engineering 84
6. Preparing diagrams and presentations in software architecture 87
7. Forming a software team to develop a project or solve a problem. 92
8. ToplantıAI — Yapay Zekâ Destekli Toplantı Yönetim Uygulaması......101
9. UML (Unified Modeling Language) diagrams......106

1. Introduction

Friends, the course I'm starting today isn't just about learning software development; it's also about how to think, plan, and shape software.

Think about constructing a building. If you don't build a solid foundation, make the walls as strong as you want, the corridors and rooms as beautiful as you want; know that even the slightest tremor won't break the building. The same applies to software: without the right architecture, without the right foundation, as the software grows, problems multiply, maintenance becomes difficult, and performance degrades.

Therefore, the focus of this course isn't just about "writing code," but about "planning the right code with the right structure." In other words, thinking like an architect.

Software design and architecture will teach you the followings:

1) Complex systems are made manageable.
2) Long-lasting, maintainable software is developed.
3) A common language can be developed when working as a team.
4) Software components written by separate individuals for different purposes function as one body when they come together for a common purpose.
5) Qualities such as performance, security, and scalability are considered from the very beginning. Risks that will arise are foreseen.
6) It does not cause any problems in updating the software, making additions, or working with other software.

Remember: A good software developer is someone who knows not only "how to write code," but also "why to write it that way." Throughout this course, I will strive to impart this perspective to you.
In the coming weeks, we will discuss real-world examples, design principles, architectural patterns, and how to structure software systems. By the end of the course, we will aim to be individuals who can not only write code but also design software architecture.

Arkadaşlar, bugün başlayacağım ders, sadece yazılım geliştirmeyi öğrenmekle ilgili değil; aynı zamanda yazılımın nasıl düşünülmesi, planlanması ve şekillendirilmesi ile ilgilidir.

Bir bina inşa etmeyi düşünün. Eğer temeli doğru yapmaz iseniz, duvarları istediğiniz gibi sağlam yapın, koridorlarını, odalarını istediğiniz güzellikte yapın; bilin ki, en ufak bir sarsıntıda bina ayakta kalmaz. Yazılımda da aynı şey geçerli: doğru mimari olmadan,

temeller doğru atılmadıkça yazılım büyüdükçe sorunlar katlanır, bakımı zorlaşır, performans düşer.

Bu yüzden bu dersin odağı, sadece 'kod yazmak' değil, 'doğru kodu doğru yapıda planlamak'. Başka bir ifadeyle, mimar gibi düşünmek.
Yazılım tasarımı ve mimarisi sizlere şunları kazandırır:

- Karmaşık sistemler yönetilebilir hale getirilir.
- Uzun ömürlü, sürdürülebilir yazılımlar geliştirilir.
- Takım halinde çalışırken ortak bir dil oluşturulur.
- Ayrı kişiler tarafından ayrı amaçlar için yazılan yazılım parçaları ortak bir amaç için bir araya geldiğinde bir vücut gibi işlev görür.
- Performans, güvenlik, ölçeklenebilirlik gibi nitelikleri daha en baştan düşünülür. Oluşacak riskler önceden öngörülür.
- Yazılım güncellemede, ilavler yapmada, başka yazılımlar ile çalışmada sorun çıkarmaz.

Unutmayın: İyi bir yazılım geliştirici, sadece 'nasıl kod yazacağını' değil, 'neden o şekilde yazması gerektiğini' bilen kişidir. Bu ders boyunca size bu bakış açısını kazandırmaya çalışacağım.
Önümüzdeki haftalarda, gerçek hayattan örnekler, tasarım prensipleri, mimari kalıplar ve yazılım sistemlerini nasıl şekillendireceğimiz üzerine konuşacağız. Dersin sonunda, sadece kod yazabilen değil, yazılımın mimarisini kurgulayabilen kişiler olmayı hedefleyeceğiz.

2. Introduction to Software Architecture

Software architecture is important because it **determines the system's success, sustainability, and ability to evolve**. Without solid architecture, even small systems become fragile, costly, and hard to maintain.

Yazılım mimarisi önemlidir çünkü sistemin başarısını, sürdürülebilirliğini ve evrimleşme yeteneğini belirler. Sağlam bir mimari olmadan, küçük sistemler bile kırılgan, maliyetli ve bakımı zor hale gelir.

Course Outline: Software Design and Architecture

Week 1 – Introduction

- Importance of software architecture and design
- Role of architects vs. developers
- Architecture, design and implementation
- Software qualities (performance, scalability, maintainability, security)

Week 2 – Software Development Life Cycle (SDLC) and Architecture

- Waterfall, Agile, DevOps perspectives on architecture
- Where architecture fits in SDLC
- Architectural decision-making process
- Case study: failure due to poor architecture

Week 3 – Software Design Principles

- Abstraction, modularity, separation of concerns
- **SOLID principles**
- Coupling and cohesion
- Design trade-offs

Week 4 – Modeling and Architectural Views

- UML basics (class diagrams, sequence diagrams, component diagrams)
- **4+1 architectural view model** (logical, development, process, physical, scenarios)
- C4 model for modern systems

Week 5 – Architectural Styles

- Layered architecture
- Client–server architecture
- Monolithic vs. microservices
- Event-driven and service-oriented architectures (SOA)

Week 6 – Architectural Patterns (Part I)

- MVC, MVP, MVVM
- Repository pattern
- Singleton, Factory, Observer, Strategy
- When to apply design vs. architectural patterns

Week 7 – Architectural Patterns (Part II)

- Microservices
- Event sourcing & CQRS
- Pipe-and-filter, blackboard, broker patterns
- Real-world examples

Week 8 – Midterm + Case Study

- Midterm exam or project checkpoint
- Case study: analyzing a real-world system's architecture (e.g., Netflix, Amazon, Spotify)

Week 9 – Quality Attributes in Architecture

- Performance, scalability, security, reliability, maintainability, usability
- Trade-offs between quality attributes
- Scenario-based evaluation (ATAM – Architecture Tradeoff Analysis Method)

Week 10 – Architectural Documentation

- Documenting architecture with UML, C4, ADRs (Architectural Decision Records)
- Communicating architecture to technical and non-technical stakeholders

Week 11 – Modern Trends

- Cloud-native architectures
- Containerization (Docker, Kubernetes)
- Serverless architectures
- API-first design

Week 12 – Architecture and Software Development Practices

- Continuous integration & continuous delivery (CI/CD)
- DevOps and architecture
- Testing in architecture (unit, integration, system tests)

Week 13 – Architecture in Large Systems

- Distributed systems
- Scalability challenges
- Fault tolerance and resilience patterns (circuit breaker, retry, bulkhead)
- Case study: global-scale web app

Week 14 – Wrap-up and Final Project Presentations

- Students present architectural designs for a chosen project
- Lessons learned in software design & architecture
- Industry best practices and career as a software architect

2.1. Importance of Software Architecture

Software architecture is **one of the most critical aspects of software engineering** because it provides the blueprint for both development and long-term evolution of a system. Its importance can be understood from several perspectives:

Yazılım mimarisi, bir sistemin hem geliştirilmesi hem de uzun vadeli evrimi için bir plan sunduğu için yazılım mühendisliğinin en kritik unsurlarından biridir. Önemi birkaç açıdan anlaşılabilir:

Purpose in the design and architecture of the software is to transform the software to be developed into a living organs. To create software that learns from collected information, an organized structure must be established. Thus, existing information sources must be brought together to establish a well-functioning architectural structure. For software that collects, classifies, and integrates information to form organisms to be integrated, the software must be able to predict and foresee its future in a balancing act. Small, scattered information and software groups must organize and learn to behave like organs, and from these organs, they must learn to behave like sentient beings.

Yazılımın tasarım ve mimarisinde amaç geliştirilecek yazılımın yaşayan bir organlara dönüştürülmesini sağlamaktır. Toplanan bilgilerden öğrenen bir yazılım oluşturabilmek için organize olabilen bir yapının oluşturulmalıdır. Böylece mevcut bilgi kaynakları bir araya getirilerek iyi işleyen bir mimari yapı kurulması gerekir. Bilgileri toplayan, sınıflandıran, bütünleştirerek organlar oluşturan yazılımın bütünleşik olabilmesi için kestirim yapabilmesi ve ve öngörüde bulunabilmesi gerekir. Küçücük dağınık bilgi yazılım grupları organize olup organ gibi davranmaya ve organlardan da hisseden canlılar gibi davranmayı öğrenmeleri gerekmektedir.

An organism hunting for information, like a food search, can classify changes to detect traces of its prey from afar. Uncertainties are numerous at this stage. To increase

accuracy when classifying changes, a research-oriented mind must be developed to recognize when more information is needed or to identify missing information. To find what it seeks within a larger mass, it must sense that it is being sought. To feel this, they must learn to work as a team. This ability to work as a team fosters the development of a problem-solving mind.

Organisms navigating through masses of information together learn to divide labor and develop a mindset focused on process management as they share the information they find. When they begin to plan their division of labor in the hunt for a larger goal, they develop a participatory mindset focused on problem solving.

In task sharing, organisms specialize in achieving success in their assigned tasks and performing them effectively, beginning to act like organs. Thus, organs are formed by sharing the functions of problem-solving. The combined action of organs creates a whole, or rather, the body.

The organs that make up the body need to develop a leader brain to sense each other, share tasks, monitor, and manage. It should be remembered that teamwork, and the ability of teams to truly feel and perceive each other, is possible through a goal-oriented, participatory mindset.

The fundamental principle of success in seizing opportunities and being different, or finding difference, is that when a team learns to achieve excellence, the mindset that recognizes the power of quality is developed. Organisms that recognize the need to be different not only in seizing threats but also in seizing opportunities must be able to focus on a single point at a time to become a team.

Organisms that quickly gather and integrate their own information, forming a body, begin to anticipate their reactions by monitoring the behavior of the sensors that are the source of this information. Rapidly perceiving the diversity and severity of the response requires the establishment of a control mechanism similar to a neural network.

Software sensors must act like organisms, distinguishing between friend and foe, and anticipating opportunities, disasters, threats, and attacks. A dataset exposed to disasters or attacks must be black-boxed to protect the information it holds up to that point. Sensors operating in neighborhoods, along with preliminary, intermediate, and centralized data collection units, must protect the information from disruptive influences.

Besin arar gibi bilgi avına çıkan organizma avının izlerini uzaktan algılaması için değişiklikleri sınıflandırması ile mümkündür. Bu aşamada belirsizliklerin sayısı oldukça fazladır. Değişiklikler sınıflandırılırken doğruluğu artırmada daha fazla bilgiye ihtiyacı olduğunu fark etmesi ya da eksik bilgiyi fark etmesi için araştırma yapmaya yönelik akıl geliştirilmelidir. Daha büyük bir yığın içerisinde aradığını bulabilmesi için aradığının arandığı hissetmesi gerekir. Hissedebilmek için ekip olmayı becermeyi öğrenmelidirler. Ekip olmayı becerebilmek problem çözmeye odaklı aklın gelişmesini sağlar.

Birlikte bilgi yığınları içerisinde dolaşan organizmalar buldukları bilgileri paylaşırken iş bölümü yapılmasını öğrenerek süreç yönetmeye yönelik aklı geliştirirler. Yazılım ekip gibi çalışmaya başlar. Daha büyük hedefi avlamada iş bölümü yapacaklarını planlamaya başladıklarında ise problem çözmeye yönelik katılımcı akıl geliştirirler.

Görev paylaşımında organizmalar üstlendikleri görevde başarılı olmak ve en iyisini yapmada uzmanlaşarak organ gibi davranmaya başlarlar. Böylece problem çözmeye yönelik işin fonksiyonlarının paylaşımı ile organlar meydana getirilmiş olur. Organların birlikte hareket etmesinden bütünlük yani bünye meydana gelir.

Bünyeyi meydana getiren organların birbirlerini hissetmeleri, görev paylaşımları, izleme, yönetme fonksiyonlarını yerine getirmesi için lider beyin oluşturmaları gerekmektedir. Başarıya giden yolda ekip olma ve ekiplerin birbirlerini çok iyi hissetmeleri ve algılamaları hedefe yönelik katılımcı akıl ile mümkün olduğu unutulmamalıdır.

Fırsatları yakalamada ve farklı olmada ya da farklılığı bulmada başarılı olmanın temel kuralı takım olarak mükemmelliği gerçekleştirmeyi öğrendiklerinde kalite gücünü fark eden aklı geliştirilmiş olur. Sadece tehditleri değil fırsatları yakalamada da farklı olmak gerektiğini hisseden organizmalar ekip olabilmek için aynı anda tek bir noktaya odaklanabilmelidirler.

Kendine ait bilgileri hızlıca toplayan, bütünleştirerek bünye meydana getiren organizmalar bilgilerin kaynağı olan algılayıcıların davranışlarını izleyerek verecekleri tepkileri önceden kestirim yapmaya başlarlar. Tepkinin farklılığı ve şiddetini hızlı algılamak için sinir ağına benzer bir kontrol mekanizmasının kurulması gerekli kılmaktadır.

Yazılımsal algılayıcıların organizma gibi davranarak dost düşman ayrımı yapması, fırsatlar, felaketleri, tehditleri ve saldırıları önceden algılaması gerekmektedir. Felaketlere ya da saldırıya maruz kalan veri yığını o ana kadar sahip olduğu bilgileri korumak için kara kutuya sahip olmalıdır. Komşuluk ilişkilerinde çalışan algılayıcılar ile

ön bilgi toplama, ara bilgi toplama ve merkezi bilgi toplama birimlerinde bilginin bozucu etkilerden korunması gerekmektedir.

2.2. Why is Software Architecture Important?

Software Architecture is the Cornerstone (Building Analogy): A software's architecture forms its framework and foundation. If the foundation is flawed, the software will fail and constantly experience problems as it grows.
Temel Köşe Taşıdır (Bina Benzetmesi): Bir yazılımın mimarisi, yazılımın çerçevesini ve temelini oluşturur. Temel kusurluysa, yazılım başarısız olur ve büyüdükçe sürekli sorunlar yaşar.

Software Architecture Manages Complexity: Large systems can have millions of lines of code. Architecture breaks down this complexity into manageable pieces.

Software Architecture Determines Quality: All quality attributes of software, such as performance, security, scalability, and maintainability, are actually derived from architecture. Good architecture = long-lasting software. (İyi mimari = uzun ömürlü yazılım.)

Software Architecture Facilitates Teamwork: When multiple teams or individuals are working on the same project, architecture provides a common language. Everyone knows which component they are responsible for.

Software Architecture Adapts to Change: The business world and technologies are constantly changing. Also software architecture is constantly changing. Software with good architecture easily adapts to new features and changes.

Software Architecture Saves Costs and Time: Architectural errors often cost a lot in the later stages of the project. Designing correctly from the beginning saves time and money in the long run.

Foundation for System Quality

- The software architecture defines how components interact, communicate, and scale each other. Because there will be anourmous lines and a lot of components of the software.
- Quality attributes such as **performance, reliability, security, maintainability, and scalability** are largely determined at the architectural level.
- A poor architecture makes software nearly impossible to patch in (yama yapmak, eklemek) quality later.

Guidance for Development Teams

- Acts as a **shared vision** of the system, aligning all developers.
- Provides a **roadmap** for implementation, ensuring different parts fit together properly.
- Reduces misunderstandings and integration problems.

Cost and Risk Management

- Good architecture helps identify and mitigate risks early (e.g., bottlenecks, technology limits).
- Prevents costly rework by making major decisions early, **before coding starts.** What should be done before starting code writing?
- Supports trade-offs between competing requirements (e.g., speed vs. security). (Rekabet eden gereksinimler arasındaki uzlaşmaları destekler.)

Enabling Change and Evolution

- Well-designed architectures allow systems to **adapt to new requirements** and technologies.
- Well-designed architectures provide modularity, making it easier to **add, replace, or upgrade** parts without breaking the whole system.
- Essential for long-lived systems that will undergo multiple versions. (Birden fazla versiyona tabi tutulacak uzun ömürlü sistemler için gereklidir.)

Communication Tool

- Good architecture serves as a **bridge between stakeholders**: developers, managers, customers, and business owners.
- **Abstract diagrams and models allow non-technical stakeholders to understand the system at a higher level.**
- Encourages informed decision-making.

In the **software engineering**, the phrase **“understand the system at a higher level”** means gaining a **broad, conceptual understanding** of how the entire system works — rather than focusing on the details of individual components or lines of code.
Here’s what that involves:

1. Looking at the big picture

You focus on the **overall structure**, **main components**, and **how they interact**.
For example:

- What are the main modules or subsystems? (What are the components of a software?)
- How do the main modules or subsystems of a sotware architecture communicate (APIs, data flow, etc.)?
- What is the main purpose or goal of the system?

2. Understanding architecture

You study the **system architecture** — that is, the design decisions that shape the system:

- Client–server structure, layers (presentation, business logic, database)
- Frameworks and technologies used
- Integration with external systems (APIs, cloud services, etc.)

3. Understanding functionality and behavior (!)

You identify what the system **does** and how users interact with software:

- What are the core use cases or user stories?
- What inputs does software take and what outputs does software produce?
- What are the system’s constraints (performance, security, scalability, manageability)?

4. Abstracting away details

At a higher level, you don’t look at specific code or algorithms.
Instead, you think in **models and diagrams**, like:

- **UML diagrams** (use case, class, sequence)
- **Data flow diagrams (DFD)**
- **Architecture diagrams**

Example: Let’s say you are analyzing an **online shopping system**:

- **Low level:** You look at code for the “Add to Cart” button.
- **High level:** You understand that the system has modules like *User Management*, *Product Catalog*, *Shopping Cart*, *Payment Gateway*, and *Order Tracking*, and they interact to fulfill user actions.

So, **understanding the system at a higher level** means seeing how everything fits together — the architecture, major components, and their relationships — without getting lost in the fine technical details.

Reuse and Productivity (Yeniden Kullanım ve Verimlilik)

- Promotes the use of **patterns and frameworks**, saving time and effort. (Desen ve çerçeve kullanımını teşvik ederek zamandan ve emekten tasarruf sağlar.)
- Components can be reused across projects if the architecture supports modularity and standard interfaces.
  (Mimari, modülerliği ve standart arayüzleri destekliyorsa bileşenler projeler arasında yeniden kullanılabilir.)
- Increases productivity by avoiding reinventing the wheel.
  (Tekerleği yeniden icat etmeyi önleyerek verimliliği artırır.)

2.3. Role of Architects and Developers

Understanding the roles of software architects and developers is crucial for both teaching and practice in software engineering. While their responsibilities overlap, the focus and scope are quite different.

Software Architect:

A **software architect** is responsible for the **big picture** of a system because their primary role is to ensure that all parts of the software work together coherently to meet both **technical** and **business goals**. (What is the software engineer's primary role?)

Here's why this is essential:

1. **System-wide vision** – The architect sees the system as a whole, not just as a collection of individual modules. They design the overall structure so that every component fits logically into the larger framework.
2. **Alignment with requirements** – The architect ensures that the software's design supports both **functional requirements** (what the system should do) and **non-functional requirements** (performance, scalability, security, maintainability, etc.).
3. **Technology decisions** – They choose the appropriate (uygun) technologies, frameworks, and design patterns that will shape how the entire system is built and maintained.
4. **Consistency and standards** – Without an architect, different developers might build modules in inconsistent ways. The architect enforces **coding standards**, **design guidelines**, and **integration strategies** to keep the system unified.
   (Mimar, sistemin birliğini sağlamak için kodlama standartlarını, tasarım yönergelerini ve entegrasyon stratejilerini uygular.)

5. **Trade-off management** – Building software involves trade-offs (e.g., performance vs. cost, flexibility vs. simplicity). The architect evaluates these trade-offs at a system-wide level to ensure long-term sustainability.
6. **Communication bridge** – The architect communicates between business stakeholders, project managers, and developers, ensuring that everyone shares a common understanding of the system's goals and structure.

In short, the architect's "big picture" focus ensures that the **system as a whole** is **coherent, scalable, and maintainable**, not just a patchwork of working parts.

A software architect is responsible for the big picture of a system. Their role is more about strategic planning, system structure, and long-term quality than writing every line of code.

Key Responsibilities:

1. Define the architecture
    - Decide on architectural style (monolith, microservices, layered, etc.).
    - Select design patterns and frameworks.
    - Establish system boundaries and module responsibilities.

2. Ensure quality attributes
    - Make trade-offs between performance, scalability, maintainability, and security.
    - Define non-functional requirements (e.g., "system must handle 1M users concurrently").
3. Technology decision-making
    - Choose languages, platforms, databases, and cloud solutions.
    - Evaluate tools, frameworks, and libraries.
4. Communicate architecture
    - Create diagrams and documentation (UML, C4 model, ADRs).
    - Explain decisions to both technical (developers) and non-technical (managers, clients) stakeholders.
5. Governance and alignment
    - Ensure different teams build components that fit together.
    - Review designs and code for architectural compliance.
    - Manage risks in scalability, security, and integration.

Software Developer:

A software developer focuses on implementation within the architectural framework. Their role is more about coding, testing, and delivering features.

Key Responsibilities:

1. Implement features
    - Write clean, maintainable, and testable code.
    - Follow coding standards and guidelines.
2. Follow architecture & design
    - Use the frameworks, patterns, and conventions defined by the architect.
    - Implement modules/components to fit the big picture.
3. Debugging and testing
    - Write unit tests, perform debugging, fix issues.
    - Ensure functionality matches requirements.
4. Collaboration
    - Work with other developers in agile teams.
    - Communicate progress, challenges, and improvements.
5. Feedback to architect
    - Report technical difficulties and suggest improvements.
    - Help refine architectural decisions based on real-world coding challenges.

Main Differences

Aspect	Architect	Developer
Scope	Entire system	Individual modules/components
Focus	Structure, strategy, quality attributes	Implementation, functionality, detail
Decisions	Frameworks, patterns, technologies	Algorithms, data structures, coding practices
Output	Architectural diagrams, guidelines, high-level designs	Source code, tests, working features
Communication	With stakeholders (business + tech)	Mostly within development team
Time horizon	Long-term evolution	Short-term feature delivery

In short:

- Architects design the blueprint and ensure system qualities.
- Developers build the house according to that blueprint.
- Both roles require collaboration — an architect without developer feedback is unrealistic, and developers without architectural guidance risk creating chaos.

2.4. Software Architecture, Design and Implementation

Software Architecture:

Software architecture is the high-level structure of a software system. It describes how the system is organized, how components interact, and how quality attributes (performance, scalability, security, maintainability, etc.) are achieved.

Key Aspects:

- Structure: Defines components (modules, services, layers) and their relationships.
- Style/Patterns: Layered, microservices, event-driven, client-server, etc.
- Technology Stack: Choice of language, database, cloud, frameworks.
- Quality Attributes: Focus on non-functional requirements like reliability, extensibility, and scalability.
- Documentation: Architectural views (UML, C4 model, 4+1 model).

Example:

An e-commerce system architect decides:

- Use microservices (for scalability).
- Each service has its own database.
- Use REST APIs for communication.
- Deploy on AWS with Kubernetes.

Software Design:

Software design is the detailed plan for how components inside the architecture will work. It operates at a lower level than architecture and bridges abstract architecture with concrete implementation.

Key Aspects:

- Design Principles: SOLID, DRY, KISS, separation of concerns.
- Design Patterns: Singleton, Factory, Observer, MVC, Repository.
- Data Structures & Algorithms: Choosing efficient methods for performance.
- Module Design: Responsibilities, interfaces, and interactions of each class/module.
- User Interface Design: Layouts, flows, and usability considerations.

Example (continuing e-commerce system):

- Cart Service: Use the Repository Pattern to manage cart data.
- Payment Service: Apply Strategy Pattern for handling multiple payment methods.
- Frontend: Implement MVC for separating UI, business logic, and data.

Software Implementation:
Software implementation is the actual coding and integration process that transforms architecture and design into a working software product.
Key Aspects:

- Coding: Writing clean, efficient, maintainable code following design specs.
- Testing: Unit tests, integration tests, end-to-end tests.
- Version Control: Using Git, branching strategies, CI/CD pipelines.
- Debugging: Identifying and fixing bugs.
- Deployment: Packaging and deploying the software in real environments.

Example (continuing e-commerce system):

- Write the CartService class in Java with Spring Boot.
- Implement a PostgreSQL repository for storing cart data.
- Write unit tests for add/remove item functions.
- Deploy the microservice in Docker containers to Kubernetes.

Comparison & Relationship:

Aspect	Architecture	Design	Implementation
Focus	System structure, high-level view	Detailed component & module design	Actual coding & integration
Abstraction level	High (macro-level)	Medium (meso-level)	Low (micro-level)
Output	Diagrams, models, guidelines	Class diagrams, design patterns, module specs	Source code, executables
Stakeholders	Architects, senior engineers, product owners	Developers, team leads	Developers, testers
Time Horizon	Long-term (system evolution)	Mid-term (project milestones)	Short-term (daily coding tasks)

How They Work Together:

1. Architecture defines the big picture (what the system should look like).
2. Design refines that picture into detailed blueprints (how modules/classes should work).
3. Implementation makes those blueprints real by writing and deploying the code.

Analogy:

- Architecture = City planning (deciding districts, roads, power lines).
- Design = Building blueprints (floor plan, wiring, plumbing).
- Implementation = Construction (laying bricks, installing systems).

In short:

- Architecture answers “What is the structure and why?”
- Design answers “How will each part work together?”
- Implementation answers “How do we code and deliver it?”

2.5. Software qualities (performance, scalability, maintainability, security)

This is one of the core topics in software architecture. In architecture, we don’t just design for functionality (what the system does), but also for qualities (how the system behaves). Here’s a detailed explanation of four fundamental software qualities:

Software Qualities:

Performance

How efficiently a system uses resources (CPU, memory, network, storage) to deliver fast responses and handle workload.

Key Metrics:

- Response time (latency)
- Throughput (transactions per second)
- Resource utilization (CPU, memory, bandwidth)

Design Considerations:

- Caching frequently used data.
- Optimizing database queries.
- Load balancing requests across servers.
- Using efficient algorithms and data structures.

Example: Google search is designed for sub-second response times, even under massive global traffic.

Scalability: The ability of a system to handle increasing workload (users, data, transactions) without performance degradation.

Types:

- Vertical scaling (scale-up): Adding more power to a single machine (e.g., faster CPU, more RAM).
- Horizontal scaling (scale-out): Adding more machines/servers to share the load.

Design Considerations:

- Stateless services (easy to distribute across servers).
- Distributed databases.
- Microservices architecture.
- Cloud-based auto-scaling.

Example: Netflix scales horizontally — when more users stream, more servers automatically spin up in the cloud.

Maintainability

How easily a system can be modified, fixed, or extended without introducing errors or breaking existing features.

Key Aspects:

- Modularity: Code organized into independent components.
- Readability: Clean, consistent coding standards.
- Testability: Easy to write automated tests.
- Documentation: Clear architecture/design docs.

Design Considerations:

- Following SOLID principles.
- Layered or modular architecture.
- Use of design patterns for common problems.

Example: A banking system may need frequent regulatory updates — if designed with modularity, only a few components need changes, not the entire system.

Security:

Protecting the system against unauthorized access, misuse, or attacks while ensuring confidentiality, integrity, and availability.

Key Aspects (CIA Triad):

- Confidentiality: Prevent unauthorized data access (encryption, authentication).
- Integrity: Prevent unauthorized data modification (checksums, digital signatures).
- Availability: Ensure the system is up and running (DDoS protection, redundancy).

Design Considerations:

- Authentication & Authorization (e.g., OAuth, role-based access control).
- Data encryption in transit (TLS) and at rest.
- Secure coding practices (avoid SQL injection, XSS).
- Regular security testing and audits.

Example: Online payment systems (e.g., PayPal) must enforce multi-factor authentication and strong encryption to protect user transactions.

Summary Table:

Quality	Focus	Example Design Decisions
Performance	Speed & responsiveness	Use caching, optimize queries, load balancing
Scalability	Growth handling	Stateless services, distributed databases, cloud auto-scaling
Maintainability	Ease of modification	Modular design, SOLID principles, good documentation
Security	Protection from threats	Encryption, authentication, secure coding practices

In short:

- Performance = Fast response under load.
- Scalability = Can grow with demand.
- Maintainability = Easy to change, fix, extend.
- Security = Safe from attacks and unauthorized access.

3. Software Development Life Cycle (SDLC) and Architecture

Software Development Life Cycle (SDLC) is a structured process defining stages of planning, creating, testing, and deploying software.
Software Architecture is the high-level structure of a software system, including its components, their interactions, and guiding principles. Software architecture provides the **blueprint (plan)** for development. SDLC provides the **process framework** for realizing that architecture. (SDLC, bu mimarinin hayata geçirilmesi için süreç çerçevesini sağlar.)
What does software architecture provide from software development?

In the **software engineering**, a **process framework** refers to a **structured set of activities, methods, practices, and guidelines** that define how software development should be planned, executed, and controlled. It serves as a **foundation** upon which specific **software process models** (like Agile, Waterfall, or Spiral) can be built and customized. (Belirli yazılım süreç modellerinin (örneğin Çevik, Şelale veya Spiral) inşa edilebileceği ve özelleştirilebileceği bir temel görevi olarak hizmet eder.)
What are specific process models in software engineering?

A **process framework** provides the **skeleton** or **blueprint** for managing and performing the software development lifecycle. It ensures consistency, quality, and efficiency across projects. (Projeler arasında tutarlılığı, kaliteyi ve verimliliği sağlar.)

Key Components of a Process Framework:

1. **Phases and Activities** – Major stages like:
    - Requirements Engineering
    - Design
    - Implementation
    - Testing
    - Deployment
    - Maintenance

    In software engineering, Requirements Engineering (RE) is the systematic process of defining, documenting, and maintaining the requirements for a software system.
2. **Roles and Responsibilities** – Define who will do what (e.g., developer, tester, project manager).
3. **Artifacts / Deliverables** – The outputs of each phase (e.g., requirement documents, design diagrams, test reports).

4. **Tools and Techniques** – Methods and technologies used (e.g., UML for design, JIRA for project tracking).
5. **Policies and Guidelines** – Rules to ensure quality and compliance (e.g., coding standards, review checklists).
6. **Metrics and Controls** – Ways to monitor progress and quality (e.g., defect rate, velocity, productivity).

Examples of Process Frameworks:

- **CMMI (Capability Maturity Model Integration)** – Focuses on process improvement and organizational maturity.
- **ISO/IEC 12207** – Defines a standard framework for software lifecycle processes.
- **Agile Frameworks** (e.g., Scrum, XP, Kanban) – Focused on iterative and incremental development.
- **RUP (Rational Unified Process)** – A customizable framework emphasizing iterative development and risk management.

Why Process Frameworks Matters:

- Provides **consistency** and **predictability** in software projects.
- Improves **communication** and **coordination** among team members.
- Ensures **quality assurance** and **process improvement**.
- Helps organizations **standardize** how projects are run.

Neden Önemlidir:

- Yazılım projelerinde tutarlılık ve öngörülebilirlik sağlar.
- Ekip üyeleri arasındaki iletişimi ve koordinasyonu geliştirir.
- Kalite güvencesini ve süreç iyileştirmesini sağlar.
- Kuruluşların projelerin nasıl yürütüleceğini standartlaştırmasına yardımcı olur.

SDLC Phases and Architectural Considerations:

1. Requirements Analysis

- Goal: Gather functional and non-functional requirements.
- Architecture Role:
  - Translate requirements into architectural drivers (performance, security, scalability).
  - Identify constraints (technology, budget, regulations).
- Architectural Output: Initial high-level architectural vision.

2. System Design

- Goal: Define how the system will be built.
- Architecture Role:
  - Establish architectural patterns (layered, microservices, event-driven).
  - Define system components, interactions, and interfaces.
  - Document using UML diagrams, C4 models, or architectural views.

- Architectural Output: Architecture document / model (logical, physical, deployment views).

3. Implementation (Coding)

- Goal: Convert design into working software.
- Architecture Role:
  - Guide developers through coding standards and frameworks.
  - Enforce modularity, separation of concerns, and adherence to the architecture.
  - Enable reusability of components.
- Architectural Output: Code that reflects the architecture.

4. Testing

- Goal: Verify system quality.
- Architecture Role:
  - Architecture should support testability (clear interfaces, modular design).
  - Ensure non-functional requirements (performance, security) can be tested.
  - Incorporate automated testing frameworks into architecture.
- Architectural Output: Architecture validated against requirements.

5. Deployment

- Goal: Release the system into production.
- Architecture Role:
  - Ensure architecture supports deployment strategies (CI/CD, containerization, cloud).
  - Design for scalability and fault tolerance.
  - Prepare for rollback and recovery mechanisms.
- Architectural Output: Deployment architecture (infrastructure as code, orchestration).

6. Maintenance & Evolution

- Goal: Fix issues, add features, adapt to changing needs.
- Architecture Role:
  - Architecture must support modifiability, extensibility, maintainability.
  - Continuous refactoring and architectural evolution.
  - Use monitoring/feedback to refine architecture.
- Architectural Output: Updated architecture to reflect system evolution.

Architectural Significance Across SDLC

- Requirements → Architecture ensures traceability from business goals to technical design.
- Design → Architecture defines patterns, quality attributes, and constraints.
- Implementation → Architecture guides developers and reduces technical debt.
- Testing → Architecture enables verification of quality attributes.
- Deployment → Architecture supports scalability, automation, and reliability.
- Maintenance → Architecture allows sustainable evolution.

SDLC Models and Architectural Implications

SDLC Model	Architectural Implication
Waterfall	Big Design Up Front, architecture fixed early.
Iterative/Agile	Architecture evolves incrementally with sprints.
DevOps/Continuous	Architecture optimized for automation, monitoring, scalability.
Spiral	Architecture refined through iterative risk analysis.
V-Model	Strong link between architecture and validation/testing.

Best Practices

- Align architecture with SDLC model chosen.
- Document architecture just enough for clarity.
- Use feedback loops (from testing, deployment, monitoring) to evolve architecture.
- Balance planned architecture and emergent design.

Conclusion

- Architecture and SDLC are interdependent:
  - SDLC provides the process roadmap.
  - Architecture ensures the system is robust, maintainable, and aligned with requirements.
- Successful projects integrate architectural thinking into every SDLC phase rather than treating it as a one-time activity.

3.1. UML for Design in Software Engineering

UML (Unified Modeling Language) is a **standard visual language** used in software engineering to **model, design, and document software systems**. It helps developers, analysts, and stakeholders **understand, visualize, and communicate** how a system will be structured and how it will behave.

UML helps whom?

Why does UML help?

Where is UML used?

Purpose of UML in Software Design

In the **design phase** of the software engineering, UML is used to:

1. **Model system architecture** before coding begins.
2. **Define components, relationships, and interactions.**
3. **Translate requirements into a concrete structure.**
4. **Serve as documentation** for future maintenance and development.

UML Diagram Categories

UML diagrams are divided into **two main types**:

1. Structural Diagrams – ***show what the system is made of***

These diagrams focus on the **static** aspects of the system — classes, objects, components, and relationships.

Diagram Type	Description
Class Diagram	Describes the classes, their attributes, methods, and relationships (inheritance, associations).
Object Diagram	Shows examples of objects and their links at a specific moment in time.
Component Diagram	Illustrates how software components or modules are organized.
Deployment Diagram	Shows how the system is physically deployed across hardware nodes.
Package Diagram	Organizes related classes or components into groups (packages).

2. Behavioral Diagrams – ***show how the system behaves***

These diagrams focus on the **dynamic** aspects — interactions, activities, and workflows.

Diagram Type	Description
Use Case Diagram	Represents system functionality from the user's perspective. Shows actors and their interactions with the system.

Diagram Type	Description
Sequence Diagram	Depicts how objects interact with each other in time sequence (message passing).
Activity Diagram	Models workflows or business processes (like flowcharts).
State Machine Diagram	Shows how an object changes states based on events.
Communication Diagram	Focuses on the relationships and messages between objects.

Benefits of Using UML in Design

- **Clarity:** Visualizes complex systems easily.
- **Consistency:** Standardized way to represent systems.
- **Communication:** Helps developers, clients, and analysts to speak a common language.
- **Early validation:** Errors in design can be found before implementation.
- **Documentation:** Provides a blueprint for development and maintenance.

Example

Imagine you're designing an **Online Shopping System**:

- **Use Case Diagram:** Shows that a *Customer* can *browse products*, *add to cart*, and *place order*.
- **Class Diagram:** Defines classes like Customer, Product, Order, and Payment.
- **Sequence Diagram:** Shows how Customer interacts with OrderService and PaymentService.
- **Deployment Diagram:** Displays the web server, database server, and client device setup.

UML in software design is a **visual modeling toolset** that bridges the gap between **requirements** and **implementation**. It helps developers **design structured, maintainable, and well-communicated systems**.

3.2. JIRA for Project Tracking in Software Engineering

JIRA (by Atlassian) is a **project management and issue-tracking tool** widely used in **software engineering** to **plan, track, and manage software development projects**. Originally created for **bug tracking**, JIRA has evolved into a **comprehensive Agile project management platform** used by developers, testers, project managers, and teams practicing **Scrum** or **Kanban**.

Purpose of JIRA in Software Engineering

JIRA helps software teams:

- **Plan** project tasks and sprints
- **Assign** responsibilities
- **Track** progress and deadlines
- **Manage** bugs, features, and improvements
- **Report** status and performance metrics

JIRA keeps the **entire software development process organized**, from idea to release.

3. Key Concepts and Features

Feature	Description
Project	A collection of issues related to one product or software effort.
Issue	The basic unit of work — can represent a *task, bug, story, or epic*.
Workflow	Defines how an issue moves through stages like *To Do* → *In Progress* → *Done*.
Board	A visual board (Scrum or Kanban) that shows issue status and progress.
Sprint	A time-boxed period (e.g., 2 weeks) in which a set of issues is completed.
Backlog	A prioritized list of work items (stories, tasks, bugs) yet to be done.
Epic	A large user story that can be broken down into smaller stories or tasks.
Story	A unit of functionality from the user's perspective (e.g., "As a user, I want to reset my password").
Bug	A problem or defect found in the software.
Dashboard & Reports	Visual summaries showing project metrics, sprint progress, burndown charts, etc.

4. How JIRA Supports Agile Development

JIRA is particularly popular in **Agile methodologies** like **Scrum** and **Kanban**:

In Scrum:

- Teams work in **sprints** (short development cycles).
- Each sprint has a defined **goal** and set of tasks.
- JIRA helps manage the **backlog**, **sprint planning**, and **daily progress**.
- Tools like **burndown charts** and **velocity reports** help track performance.

  What is sprints in Scrum?

In Kanban:

- Work is visualized on a **Kanban board**.
- Issues move through columns (e.g., *To Do → In Progress → Testing → Done*).
- Focuses on **continuous delivery** and **limiting work in progress (WIP)**.

5. Example Workflow

Imagine a **software team** developing a new mobile app:

1. **Product Owner** creates user stories in JIRA:
   - "As a user, I want to log in using Google."
   - "As a user, I want to save my favorite items."
2. **Developers** and **testers** estimate and assign tasks.
3. The team starts a **2-week sprint** using a Scrum board.
4. Each task moves across the workflow:
   To Do → In Progress → Code Review → Testing → Done
5. **JIRA dashboards** show:
   - Sprint progress (burndown chart)
   - Number of completed tasks
   - Remaining bugs or blockers

6. Benefits of Using JIRA

✅ **Centralized management** — everything in one place

✅ **Transparency** — the entire team can see progress

✅ **Collaboration** — comments, attachments, and mentions keep communication clear

✅ **Customization** — adaptable workflows for any project style

✅ **Integration** — works well with GitHub, Bitbucket, Confluence, and CI/CD tools

7. Example JIRA Board (Conceptually)

To Do	In Progress	Code Review	Done
User login screen	Implement API	Bug fix #223	UI finalized
Shopping cart test			

JIRA is a powerful project-tracking and collaboration tool that helps software teams **organize work, follow progress, and deliver high-quality software efficiently.**

3.3. Waterfall, Agile, and DevOps Perspectives on Architecture

Software development methodologies influence how software architecture is approached, designed, and evolved. The architecture perspective differs significantly across Waterfall, Agile, and DevOps, shaping project success, quality, and adaptability.

(Yazılım geliştirme metodolojileri, yazılım mimarisine nasıl yaklaşıldığını, tasarlandığını ve geliştirildiğini etkiler. Mimari bakış açısı, Waterfall, Agile ve DevOps'ta önemli ölçüde farklılık göstererek proje başarısını, kalitesini ve uyarlanabilirliğini şekillendirir.)

Agile Development, yazılım geliştirme sürecinde esneklik, hızlı geri bildirim, sürekli iyileştirme ve müşteri memnuniyetini esas alan bir yöntemdir. Klasik (Waterfall) modele göre daha dinamik ve tekrarlamalı (iteratif) çalışır.

Waterfall Development (Şelale Modeli), yazılım geliştirme sürecinin ardışık aşamalardan oluştuğu, her aşamanın tamamlanmadan bir sonrakine geçilemediği doğrusal (linear) bir modeldir. Yani süreç bir şelale gibi yukarıdan aşağıya doğru akar.

Waterfall Perspective on Architecture:

Characteristics

- Sequential, phase-based development (requirements → design → implementation → testing → deployment).
- Architecture is defined early in the project and remains relatively fixed.

Architecture Approach

- Big Design Up Front (BDUF): Comprehensive architectural design before coding starts.
- Detailed documentation and system blueprints prepared at the start.
- Emphasis on predictability, control, and traceability.

Benefits

- Clear vision of the entire system.
- Easier to manage in large, stable environments (e.g., government, defense).
- Well-suited for projects with stable, unchanging requirements.

Challenges

- Inflexibility: Hard to adapt architecture if requirements change.
- Risk of outdated design by the time system is implemented.
- Long feedback cycles → architectural mistakes discovered late.

Agile Perspective on Architecture:

Characteristics

- Iterative and incremental development.
- Architecture evolves alongside software features.
- Focus on responding to change and delivering value and report quickly.

Architecture Approach

- Emergent Architecture: High-level guidelines created initially, but detailed architecture evolves through sprints.
- Architecture supports continuous refactoring.
- Collaboration between developers and architects (architect as a facilitator, not a dictator). (Geliştiriciler ve mimarlar arasında işbirliği (mimarın diktatör değil, kolaylaştırıcı olması).)

Benefits

- Flexibility to adapt architecture as requirements evolve.
- Faster delivery of working features.
- Encourages lightweight documentation and cross-functional teams.

Challenges

- Risk of architectural drift or erosion without long-term vision.
- Potential technical debt if architectural decisions are postponed too much.
- Requires experienced teams to balance emergent vs. planned architecture.

DevOps Perspective on Architecture:

Characteristics

- Extends Agile principles by focusing on continuous delivery, deployment, and operations.
- Strong collaboration between development and operations.
- Architecture must support automation, monitoring, and scalability.

Architecture Approach

- Architecture as Code: Infrastructure and deployment pipelines automated.
- Systems designed for observability, resilience, and scalability (e.g., microservices, cloud-native).
- Emphasis on feedback loops from production to improve design.

Benefits

- Continuous feedback ensures architecture aligns with real-world usage.
- Supports scalable, fault-tolerant systems.
- Automation reduces human error and accelerates delivery.

Challenges

- Requires significant cultural and organizational change.
- More complex architectures (e.g., microservices) introduce integration and monitoring challenges.
- Demands strong tooling and cloud infrastructure expertise.

Comparative View

Aspect	Waterfall	Agile	DevOps
Architecture timing	Early, fixed (BDUF)	Emergent, evolving	Continuous, production-driven
Flexibility	Low	Medium to high	High
Documentation	Heavy, detailed upfront	Lightweight, evolving	Automated + monitored
Risk	Late discovery of design flaws	Technical debt if unmanaged	Operational complexity
Best for	Stable, regulated environments	Dynamic, evolving requirements	Fast delivery, scalable cloud-native

- Waterfall: Stable and predictable, but rigid.
- Agile: Flexible and adaptive, but needs strong governance to avoid chaos.
- DevOps: Continuous improvement and scalability, but operationally complex.

Modern organizations often combine Agile + DevOps approaches while borrowing some Waterfall rigor for compliance-heavy domains.

3.4. Where Architecture Fits in SDLC (Software Development Life Cycle)

Software Development Life Cycle (SDLC) is a structured process defining stages of planning, creating, testing, and deploying of a software.
Software Architecture is the high-level structure of a software system, including its components, their interactions, and guiding principles. Software architecture provides the blueprint (plan) for development. SDLC provides the process framework for realizing that architecture. (SDLC, bu mimarinin hayata geçirilmesi için süreç çerçevesini sağlar.)

Software Development Life Cycle:

1. Requirement Analysis Phase
2. System Design Phase
3. Implementation Phase
4. Testing Phase
5. Deployment Phase
6. Maintenance & Evolution Phase

1. Requirement Analysis Phase: Architecture **does not start here**, but requirements **drive architecture**. Functional and non-functional requirements (performance, scalability, security, maintainability, etc.) set the **architectural quality goals**. Example: If the requirement is *high availability*, the architecture must support redundancy and fault tolerance.

Architectural quality goals are measurable, prioritized expectations about a software system's performance, behavior, and constraints. They ensure the architecture supports business needs, user expectations, and long-term sustainability. (Mimari kalite hedefleri, bir yazılım sisteminin performansı, davranışı ve kısıtlamaları hakkında ölçülebilir, önceliklendirilmiş beklentilerdir. Mimarinin iş ihtiyaçlarını, kullanıcı beklentilerini ve uzun vadeli sürdürülebilirliği desteklemesini sağlarlar.)

Common Categories of Architectural Quality Goals:

- Performance
- Scalability
- Availability
- Reliability
- Maintainability
- Testability
- Security
- Usability

- Interoperability
- Portability
- Extensibility
- Modifiability
- Observability

(Performans, Ölçeklenebilirlik, Kullanılabilirlik, Güvenilirlik, Bakım Kolaylığı, Test Edilebilirlik, Güvenlik, Kullanılabilirlik, Birlikte Çalışabilirlik, Taşınabilirlik, Genişletilebilirlik, Değiştirilebilirlik, Gözlemlenebilirlik)

Why Architectural Quality Goals Matter?
Because architecture is **all about trade-offs**.
For example:

- To increase **performance**, you may sacrifice **modifiability**.
- To increase **security**, you may reduce **usability**.
- To increase **availability**, you increase **cost**.

Quality goals make these trade-offs explicit and aligned with business priorities.

2. System Design Phase
This is where **software architecture is most prominent**.
Architects define:

  - **System structure** (layers, components, subsystems)
  - **Interactions** (APIs, communication protocols, data flow)
  - **Technology stack** (databases, frameworks, cloud services)
  - **Architectural patterns** (layered, microservices, event-driven, etc.)
- The output: **Architectural blueprints, diagrams, and guidelines** that developers will follow.

3. Implementation Phase
Developers implement code **according to the architecture**. Architecture provides **constraints and guidance**:

  - Which design patterns to use.
  - How modules should interact.
  - What are the coding standards and integration rules?

4. Testing Phase
Architecture influences **test strategy**: Unit, integration, system, and performance testing depend on architectural design. Example: A microservices architecture requires testing service contracts and API gateways.

5. Deployment Phase

Architecture decisions (e.g., monolith vs. cloud-native microservices) affect **how software is deployed**:

- CI/CD pipelines
- Container orchestration (Docker, Kubernetes)
- Infrastructure as Code

6. Maintenance & Evolution Phase

Architecture determines **ease of maintenance and scalability**.

- A well-designed architecture:
  - Simplifies bug fixing
  - Enables smooth feature addition
  - Reduces technical debt
- Poor architecture: leads to high maintenance costs, scalability bottlenecks.

Summary

- **Requirements** → guide architecture
- **Design** → architecture is defined
- **Implementation, Testing, Deployment, Maintenance** → architecture is followed, validated, and evolved

So, **architecture fits mainly in the *Design Phase*** of SDLC, but its influence extends across **all phases**, ensuring the system's quality attributes (performance, scalability, security, etc.) are met.

3.5. Architectural Decision-Making Process

Architecture isn't just about drawing diagrams — it's about making **informed trade-offs**. The process ensures that architectural choices align with business goals, technical constraints, and quality attributes.

Business goals describe *why* the system is being built from a business perspective. They define the **purpose, business value, priorities, and expected outcomes** of the software. T**hey focus on:**

- Market needs
- Customer expectations
- Revenue models
- Competitive advantage
- Regulatory or organizational needs

Technical constraints are limitations or conditions imposed on the system that restrict architectural choices. They are non-negotiable boundaries. These may include:

- Required programming languages or frameworks
- Existing legacy systems
- Cloud/on-premise restrictions
- Hardware limitations
- Budget and timeline limits
- Compliance and security standards
- Organizational technology standards

Quality attributes (also called architectural quality goals or non-functional requirements) describe how well the system should perform. They determine important runtime and developmental characteristics.
Common categories: Performance (latency, throughput), Scalability, Availability, Security,
Maintainability, Reliability, Testability, Usability, Interoperability, Portability, Extensibility, Observability

1. Identify Requirements

- **Inputs:**
  - Functional requirements (what the system must do)
  - Non-functional requirements (performance, scalability, security, reliability, etc.)
  - Business goals, budget, and timelines

2. Define Quality Attributes

- Quality attributes drive architectural choices:
  - **Performance**
  - **Scalability**
  - **Availability**
  - **Maintainability**
  - **Security**
- Example: A banking system prioritizes *security and consistency*, while a streaming platform prioritizes *scalability and performance*.

3. Explore Alternatives

Identify candidate architectural styles/patterns: Monolithic vs. Microservices, Layered vs. Event-driven, On-premises vs. Cloud-native. Each alternative is evaluated against requirements and constraints.

4. Evaluate Trade-offs

Use methods like: **ATAM (Architecture Tradeoff Analysis Method), Cost-benefit analysis, Risk assessment.** Example: Microservices offer scalability but add complexity in deployment and monitoring.

ATAM (Architecture Tradeoff Analysis Method) is a **systematic, scenario-based, stakeholder-driven method** used to **evaluate a software architecture** by analyzing how well it satisfies **quality attributes** (performance, security, modifiability, availability, etc.) and understanding the **trade-offs** the architecture makes. Developed by the Software Engineering Institute (SEI) at Carnegie Mellon University, ATAM is one of the most widely used architecture evaluation techniques in industry and academia.

ATAM is a **structured process** used during early architecture design to:

- Identify **architectural risks**
- Uncover **trade-offs** between quality attributes
- Evaluate whether architecture decisions support **business goals**
- Provide stakeholders with a shared, technical understanding of the system

ATAM is *not* about validating functionality—it's about validating **how good the architecture is** using quality drivers.
ATAM focuses on i**dentifying quality attribute trade-offs** e.g., improving performance may reduce modifiability, increasing security may reduce usability.
Revealing sensitivity points: Design decisions where small changes create large architecture impacts.
Revealing risk points: Areas where quality goals may not be achieved.
Revealing non-risks: Areas where architecture clearly fulfills the quality goals.

In software design and architecture, **“understanding the trade-offs the architecture makes”** means recognizing that every architectural decision **improves some qualities** of the system while often **reducing or limiting others**. Architecture is fundamentally about **balancing competing quality attributes**, not maximizing all of them at the same time. A **trade-off** is a situation where improving one quality attribute *negatively affects* another, due to limited resources, constraints, or inherent design characteristics.

Architecture is always a negotiation between:

- Business goals
- Technical constraints
- Quality attributes
- Cost & schedule
- Team skill set
- Available technologies

Because these factors rarely align perfectly, architects must make **choices** that inevitably involve compromises.

Ensuring alignment with business goals: Architecture must support what the business values most (e.g., performance > modifiability).

5. Make and Document Decisions choose the **best-fit architecture** considering trade-offs.

Document: Decision rationale, Alternatives considered, Trade-offs made, Impact on system qualities. This creates **Architectural Decision Records (ADRs)** for traceability.

Architectural Decision Records (ADRs) are **small, structured documents** that capture important architectural decisions, along with the context, reasoning, and consequences behind those decisions. They ensure that every significant design choice is **explicit**, **traceable**, and **understandable in the future**. ADRs are widely used in modern software engineering because they help teams avoid confusion, knowledge loss, and undocumented decisions.

An **ADR** documents:

1. **What decision was made**
2. **Why it was made**
3. **The alternatives considered**
4. **The consequences of the decision**
5. **When and by whom**

6. Validate Decisions

- Prototype critical parts ("spikes") to validate feasibility.
- Conduct **proof-of-concept (PoC)** or **performance benchmarking**.
- Gather stakeholder feedback.

In software design and architecture, **"conduct proof-of-concept (PoC) or performance benchmarking"** means creating a **small, targeted experimental implementation** to test whether a specific architectural idea, technology, or component **will actually work** in the real system—before committing time, money, and full-scale development. This is one of the most important techniques architects use to **reduce risk**, validate decisions, and compare alternatives.

A **Proof-of-Concept** is a **minimal, experimental implementation** built to test whether an architectural idea is feasible.

Purpose of a PoC:

- Validate a risky architectural assumption
- Check if a technology is suitable
- Explore unknowns (performance, integration, scalability, security)
- Prove that something *can* be built

A PoC is not a production-ready system; it is a quick experiment.

Performance benchmarking means **measuring and comparing the performance** of different architectural approaches, technologies, or configurations under realistic conditions.

Typical benchmarks measure:

- Response time
- Throughput
- Latency
- Scalability
- Memory usage
- CPU consumption
- Disk I/O
- Network overhead

Why benchmark? Performance depends on real-world behavior, not assumptions. Benchmarking allows teams to choose the best architecture **based on evidence**.

Why Architects Do PoCs and Benchmarking: Because many architectural decisions involve **uncertainty**.

7. Communicate & Align

Share architecture decisions with: Developers, Testers, Operations teams, Business stakeholders. Ensure everyone understands **why** certain decisions were made.

8. Monitor & Revisit

Architecture is not static. Regularly review decisions as: Requirements evolve, Technology changes, New risks emerge, Adjust architecture while minimizing technical debt.

The **architectural decision-making process** can be seen as a cycle:

1. Gather requirements
2. Define quality attributes
3. Explore alternatives
4. Evaluate trade-offs
5. Decide & document (ADRs)
6. Validate via PoC/prototyping
7. Communicate & align
8. Monitor & revisit

4. Software Design Principles

Software design principles aim to **reduce complexity, improve readability, and make systems easier to maintain and extend**. They are applied during the **architecture and design phases** of SDLC (**Software Development Life Cycle)**.

1. Single Responsibility Principle (SRP)

A class/module should have **only one reason to change**. **Goal is e**ach component focuses on a single functionality or concern.

- **Example:**
  - A UserManager class handles user logic only.
  - Logging should be in a separate Logger class.

A function is the smallest unit of behavior in a program. A function is a block of code that performs one specific task and can be called repeatedly. Function does a single task.
Function should do one thing and SRP applies. Function should have clean inputs and outputs. Function helps you avoid code duplication.

A class is a blueprint (plan) for creating objects. Class bundles data (attributes) and behavior (methods) together. *(Sınıf, verileri (öznitelikleri) ve davranışları (yöntemleri) bir araya getirir.)* A class represents something meaningful in your system (e.g., User, Order, Invoice). **A class, module, or function should have *one and only one* reason to change.** Class represents a thing with data and behavior. A class should have one responsibility (SRP). A class should depend on abstractions, not details (DIP). A class helps group related behaviors/data.

A module is a file (or a group of files) that organizes related classes and functions. Module organizes related classes/functions into a structure. A module organizes code logically. A module should group elements with related responsibilities. Modules should be loosely coupled and highly cohesive.

SRP does *not* simply mean "do one thing." It means the module should have **one responsibility** — **one job**, **one purpose**, and therefore **one reason to change**. A *responsibility* usually means a **role** in the system.

Why is SRP important?
Easier Maintenance: If a class has one responsibility, changing that behavior does not affect other unrelated behaviors.
Lower Coupling: SRP reduces the chance that a change in one part of the system will break another part.

Better Readability: Classes/functions become more understandable — you instantly know what they do.
Better Testability: Small, focused classes/functions are easy to write unit tests for.

What happens when SRP is violated? (**SRP ihlal edildiğinde ne olur?)**
A class does too many things. Examples: A UserService that manages user data, performs logging and sends emails. A ReportGenerator that fetches data, processes data, writes files to disk. Such classes become **God classes**, difficult to maintain and fragile to change.

2. Open/Closed Principle (OCP)

Software entities (classes, modules, functions) should be **open for extension but closed for modification**. **Goal is to add** new features without altering existing code, reducing risk of bugs. **Example:** Using **interfaces or abstract classes** so new functionality can be added through inheritance or composition.

3. Liskov Substitution Principle (LSP)

Subtypes must be **substitutable for their base types** without altering correctness. **Goal is to m**aintain **behavioral consistency** in inheritance hierarchies. **Example:** If Bird has a method fly(), a subclass Penguin should not break the behavior expected from Bird. (Better design: separate flying ability.)

4. Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they **don't use**. **Goal is to a**void "fat" interfaces that mix unrelated functionality. **Example:** Split Worker interface into IWorkable and IEatable instead of one big interface with unrelated methods.

5. Dependency Inversion Principle (DIP)

High-level modules should **not depend on low-level modules**; both should depend on abstractions. **Goal is to r**educe tight coupling and improve flexibility. **Example:** PaymentProcessor depends on an IPaymentGateway interface instead of a concrete PaypalGateway.

6. Don't Repeat Yourself (DRY)

Avoid **duplicating code or logic**. **Goal is** Centralize knowledge to reduce maintenance errors. **Example:** Extract reusable utility functions instead of copying code in multiple places.

The **Don't Repeat Yourself (DRY)** principle is one of the most fundamental and widely applied software design principles. **DRY means that every piece of knowledge, logic,**

or behavior should exist in your system in one—and only one—place. In other words, a**void duplication of code, logic, and knowledge.**

Why DRY is important

- **Easier maintenance:** If business rules live in one place, and the rule changes, you only update *one* part of the code — not many copies.
- **Fewer bugs:** Duplicated code often becomes inconsistent over time. One copy gets updated, another doesn't **bug**.
- **Cleaner code:** DRY produces more modular code, reusable components, separation of concerns.

What counts as repetition?

- **Copy-pasting code:** If two functions have the *same or similar* logic, that's duplication.
- **Duplicated business rules:** If "calculate tax = 0.18" exists in 5 places. There is duplication.
- **Duplicated knowledge:** If both OrderService and InvoiceService validate email format. This is duplication.

Every piece of logic should have a single, unambiguous, authoritative source.

7. Keep It Simple, Stupid (KISS)

Systems should be **as simple as possible**, but no simpler. **Goal is to a**void unnecessary complexity. **Example:** Avoid convoluted one-liner code that's hard to understand; prefer readable loops and clear logic.

8. YAGNI (You Aren't Gonna Need It)

Don't implement features **until they are actually needed**. **Goal is to p**revent over-engineering and wasted effort. **Example:** Don't build a complex reporting system if the client hasn't requested it yet.

9. Separation of Concerns (SoC)

Different concerns should be **handled by separate modules**.**Goal is m**odular, maintainable, and testable code. **Example:** MVC pattern (Model-View-Controller): Model is data, View is UI and Controller is business logic.

Separation of Concerns (SoC) is one of the most important high-level design principles in software engineering. It guides how you **organize and structure your system** so that each part handles a **distinct concern**.

Separation of Concerns means dividing a software system into distinct sections, where each section is responsible for a clearly defined concern (task, responsibility, or aspect).

A **concern** = something your code is responsible for, such as business logic, user interface, data Access, logging, error handling, security and configuration. Each should be separated, not mixed into one place.

Why SoC matters

- **Makes code easier to understand:** You know exactly where a specific part of the behavior lives.
- **Easier maintenance:** Fixing or updating one concern doesn't break others.
- **Reusability:** Separated modules can be reused across applications.
- **Easier testing:** Each concern can be tested independently.

SoC vs SRP (They are related but not the same!)

Principle	Scope	Meaning
SRP	Class/function level	A class should have one reason to change
SoC	System/module level	Divide the system into separate parts with different concerns

SRP = small scale

SoC = big scale

Real-world examples of SoC

- MVC (Model–View–Controller)
- 3-tier architecture:
  - Presentation layer
  - Business layer
  - Data access layer
- Microservices (each service is a concern)
- CSS vs HTML vs JavaScript (web SoC)

SoC in one sentence: "Keep different things separate so each part of the system deals with only one aspect of the problem."

10. Principle of Least Astonishment

Software should **behave in a way that users or developers expect**. **Goal is to m**ake software predictable and intuitive. **Example:** A function named deleteUser() should **actually delete a user**, not just deactivate it silently.

Summary Table

Principle	Goal
SRP	One class, one reason to change
OCP	Extend without modifying existing code
LSP	Subtypes substitute base types correctly
ISP	Avoid forcing clients to use unused methods
DIP	Depend on abstractions, not concretions
DRY	Reduce code duplication
KISS	Keep design simple
YAGNI	Don't add unused features
SoC	Separate independent concerns
Least Astonishment	Predictable behavior

These principles **work together**: SRP supports OCP and SoC; DIP reduces coupling, aiding maintainability; DRY + KISS reduce complexity.

4.1. Abstraction, Modularity, and Separation of Concerns (SoC)

These three concepts—**Abstraction, Modularity, and Separation of Concerns (SoC)**—are foundational in software design and architecture. They often overlap but have distinct roles in creating maintainable and scalable systems. Let's break them down clearly.

1. Abstraction

Definition

- Abstraction is the process of **hiding the internal details** of a component or system while exposing only the necessary functionality.
- Think of it as a **simplified view** that lets users or developers interact with a system without worrying about its inner workings.

Goal

- Reduce complexity
- Focus on **what a system does** rather than **how it does it**

Examples

1. **Programming:** A List interface in Java exposes methods like add() or remove() without revealing the underlying implementation (array, linked list, etc.).
2. **Real life:** Driving a car—you use the steering wheel and pedals without knowing how the engine works.

2. Modularity

Definition

- Modularity is the practice of **dividing a software system into separate, independent modules** that can be developed, tested, and maintained independently.

Goal

- Improve maintainability, readability, and reusability
- Allow **parallel development** by multiple teams

Key Features of Modules

- Encapsulation of functionality
- Clear **interfaces** for interaction with other modules
- Low coupling (independent modules)
- High cohesion (internally focused functionality)

Examples

1. In a web application:
    - UserModule handles user management
    - PaymentModule handles payments
    - NotificationModule handles emails/SMS
2. Each module can be **developed or replaced independently**.

3. Separation of Concerns (SoC)

Definition

- SoC is the practice of **dividing a system so that each part addresses a separate concern** or responsibility.
- Concern = a specific aspect of the system (e.g., UI, business logic, data access)

Goal

- Reduce complexity
- Make systems easier to understand, maintain, and extend
- Promote **modularity** (SoC often drives modular design)

Examples

1. **MVC Pattern:**
    - Model → Data and business logic
    - View → User interface
    - Controller → Handles input and application flow
2. Web application layers:
    - Presentation layer, Service layer, Data Access layer

◆ **How They Relate**

Concept	**Focus**	**Example**
Abstraction	Hiding complexity, showing only essential features	Java List interface
Modularity	Structuring system into independent components	UserModule, PaymentModule
Separation of Concerns	Ensuring different responsibilities don't mix	MVC pattern, Layered architecture

Relationship:

- SoC drives **modularity**: each module handles one concern.
- Abstraction is used **within modules** to hide internal details.

Summary:

- **Abstraction:** What it does, not how it does it.
- **Modularity:** Break system into independent, replaceable modules.
- **Separation of Concerns:** Keep different responsibilities separate.

5. Creating a Software Architecture

Creating a software architecture essentially means planning the "fundamental framework of the software." This process takes into account not only technical decisions but also business objectives, user needs, and the long-term evolution of the system.

In the problem solving, the software engineer either develops the software architecture himself or comes to the project with a pre-prepared one. In the software engineering, the software development process is phased according to the pre-prepared software architecture. The assigned software engineers are organized to allow for teamwork. They either determine who works where or are given options.

How to Create a Software Architecture?

1. Defining Requirements

- Functional requirements: What will the system and the software do? (For example: product listing, shopping cart, payment processing on an e-commerce site).
- Non-functional requirements: Criteria that determine software quality, such as performance, security, scalability, usability, and flexibility.
- Constraints: Budget, time, platform/technology to be used, compatibility requirements.

2. Prioritizing Quality Features

- Is performance more critical, security, or ease of maintenance?
- For example: Security is more important in banking systems, and scalability is more important in social media applications.
- These priorities directly influence the technical decisions to be made in the architecture.

3. Selecting the Architectural Style or Approach

- Monolithic Architecture: For simple, single-piece applications.
- Layered Architecture: A structure divided into layers, such as presentation, business logic, and data access.
- Microservices: Systems composed of large, scalable, and independent services.
- Event-Driven: Asynchronous, messaging-oriented systems.
- Serverless / Cloud-Based: Cloud services that scale according to need.

4. Defining Architectural Components

- Modules, services, databases, user interfaces.
- Determining the responsibilities of these components and how they will communicate with each other.
- For example: API communication, event bus usage, data sharing.

5. Selecting Technologies and Tools

- Programming language, framework, database type (SQL/NoSQL), communication protocols (REST, gRPC, GraphQL).
- Cloud platforms (AWS, Azure, GCP) or on-premises options.

6. Preparing Architectural Diagrams

- High-level diagrams: System components and their relationships.
- Detailed diagrams: Data flow, API communications, deployment diagrams.
- Possible notation: UML, C4 model.

7. Prototype / Proof of Concept Development

- A small prototype to test the feasibility of architectural decisions.
- For example: If a microservice is selected, try it on a service.

8. Documenting Architectural Decisions

- Which technology was chosen and why?
- Which quality features were prioritized?
- Architectural decisions can be written in the Architecture Decision Record (ADR) format.

9. Obtaining Feedback and Revision

- Evaluation with the development team, business unit, and security experts.
- Updating the architecture if there are any deficiencies.

10. Evolution and Continuous Improvement

- Architecture is not an immutable (değişmez) plan. It must be updated as business needs, user numbers, and technology trends change.

5.1. Defining Requirements in Software Engineering

In software engineering, **defining requirements** means **identifying, analyzing, documenting, and validating what a software system must do and how it should perform**.

These requirements describe both: **Functional requirements** — *what the system should do*, and **Non-functional requirements** — *how well it should perform*. In short: Defining requirements is the process of turning user needs and business goals into clear, testable, and agreed-upon specifications.

Importance of Defining Requirements

Requirements definition is the **foundation of successful software development**.
It ensures that all stakeholders — users, developers, testers, and managers — share a **common understanding** of what is being built.
If requirements are poorly defined, it often leads to:

- Misunderstandings between the client and the development team
- Unnecessary rework and delays
- Cost overruns and project failure

A well-defined set of requirements ensures:
✅ Correctness — The software meets real user needs
✅ Clarity — Everyone understands what's expected
✅ Traceability — Every feature can be linked to a requirement
✅ Measurability — Requirements can be tested and verified

Types of Software Requirements

Category	Description	Examples
Functional Requirements	Specify what the system should do — tasks, functions, or operations.	User login, report generation, data entry, search functionality
Non-Functional Requirements (Quality Attributes)	Define how the system performs — qualities and constraints.	Performance, security, usability, scalability
Business Requirements	Describe high-level goals and objectives of the organization.	“Increase online sales by 20% in one year.”
User Requirements	Express needs from the user’s perspective (often in natural language).	“As a user, I want to reset my password if I forget it.”
System Requirements	Detailed technical descriptions of system behavior and interfaces.	Database schemas, API endpoints, network configurations
Regulatory Requirements	Requirements based on legal or industry standards.	GDPR compliance, ISO/IEC standards

The Requirements Definition Process

Defining requirements typically follows these main steps:

Step 1: Requirements Elicitation (işbirliği)

- Gather information about what users and stakeholders need.
- Techniques studies: interviews, surveys, workshops, brainstorming, observation, document analysis.
- Goal: capture both **explicit** and **implicit** needs (hem açık hem de örtük ihtiyaçları yakalanır).

Step 2: Requirements Analysis

Examine and refine gathered requirements:

- Remove contradictions or ambiguities (Çelişkileri veya belirsizlikleri ortadan kaldırın).
- Check for feasibility, completeness, and consistency (Uygulanabilirliği, bütünlüğü ve tutarlılığı kontrol edin).
- Prioritize based on business value and constraints (İş değerine ve kısıtlamalara göre önceliklendirin).

Step 3: Requirements Specification
Document the requirements formally, typically in a **Software Requirements Specification (SRS)** document.
This includes:

- System objectives
- Functional and non-functional requirements
- Use cases and user stories
- Constraints and assumptions

Step 4: Requirements Validation
Ensure that documented requirements accurately represent stakeholder needs.
Techniques: reviews, prototypes, simulations, and walkthroughs.
Step 5: Requirements Management
Track and update requirements throughout the project lifecycle, especially when business needs or technologies change.

Characteristics of Good Requirements
Good requirements should be:

Characteristic	Explanation
Correct	Accurately describes system needs
Unambiguous	Clearly stated, only one interpretation possible
Complete	All necessary requirements are included
Consistent	No contradictions between requirements
Verifiable	Can be tested or measured objectively
Feasible	Realistic within time, cost, and technical constraints
Traceable	Each requirement can be linked to its origin
Prioritized	Ranked by importance or business value

Common tools used during requirement definition include:
Jira, **Trello**, or **Asana** for tracking user stories and issues
IBM Rational DOORS, **ReqView**, or **Helix RM** for formal requirement management
Use Case Diagrams and **User Stories** for visual and narrative representation
Prototyping tools (like Figma or Balsamiq) for user interface requirements

Challenges in Defining Requirements

- Ambiguous or changing stakeholder needs
- Communication gaps between business and technical teams
- Overlooking non-functional requirements
- Scope creep (new requests added continuously)

- These challenges highlight the need for **clear documentation, active communication, and validation** throughout the project.

Relationship to Software Architecture

Defined requirements directly influence **architectural design**:

- High performance may require distributed systems
- High security may require layered architecture and encryption
- Frequent updates may suggest microservices or modular design
- Architecture is shaped by the requirements — it's the structure built to satisfy them.

Conclusion

Defining requirements is one of the most crucial steps in software engineering. It transforms stakeholder needs into actionable specifications that guide design, implementation, and testing.

Well-defined requirements = clear vision, fewer surprises, and a higher chance of project success.

5.2. Prioritizing Quality Features in Software Engineering

In software engineering, **prioritizing quality features** means **identifying which quality attributes (also called non-functional requirements)** are the most important for a specific software system and ensuring that design and implementation decisions support them.

These **quality features** determine **how well** the system performs its functions — not *what* it does, but *how* it does it.

In short: Prioritizing quality features means deciding *which system qualities matter most* (like performance, security, usability, or scalability) and allocating resources to achieve them.

Functional vs. Non-Functional Requirements

To understand quality features, we must distinguish between two types of requirements:

Type	Description	Examples
Functional Requirements	Describe *what* the system should do.	User login, order processing, data storage
Non-Functional Requirements (Quality Features)	Describe *how well* the system performs its functions.	Reliability, performance, usability, security

Prioritizing quality features focuses on non-functional requirements.

Common Software Quality Features

According to ISO/IEC 25010 (Software Quality Model), important quality attributes include:

Quality Feature	Description
Performance Efficiency	The system's speed and resource usage under specific conditions.
Reliability	The ability to operate consistently and recover from failures.
Scalability	Capacity to handle growing amounts of work or users.
Security	Protection against unauthorized access or attacks.
Usability	How easy it is for users to learn and operate the system.
Maintainability	How easily the system can be modified, updated, or debugged.
Portability	The ease of running the software on different environments.
Compatibility	Ability to work with other systems or software.

Why Prioritization is Necessary

In real-world projects, it's **impossible to optimize all quality features equally** — improving one often impacts another.

For example:

- Increasing **security** might reduce **performance**.
- Improving **usability** could increase **development time**.
- Maximizing **scalability** might raise **infrastructure costs**.

Therefore, software teams must **prioritize** based on:

- Business goals
- User expectations
- Technical constraints
- Budget and time limits

5. How to Prioritize Quality Features

Step 1: Identify Stakeholders and Their Needs

Gather input from:

- Clients and users (what they value most)
- Developers and architects (technical feasibility)
- Business managers (strategic goals)

Step 2: Define Quality Scenarios

Create concrete situations to describe how a quality feature will be evaluated.
Example: "The system should handle 5,000 simultaneous users without performance degradation."

Step 3: Use Prioritization Techniques

Several frameworks help teams rank quality attributes:

Technique	Description
MoSCoW Method	Classify features as Must have, Should have, Could have, or Won't have.
QFD (Quality Function Deployment)	Links customer needs to design priorities.
AHP (Analytic Hierarchy Process)	Compares attributes pairwise to determine relative importance.
Weighted Scoring	Assigns numerical weights to each quality attribute.

Step 4: Document and Communicate

Record priorities in the **Software Requirements Specification (SRS)** or **Architecture Decision Record (ADR)** and communicate them clearly to all technical teams.

Step 5: Reflect Priorities in Design

Architectural decisions should reflect the chosen priorities:

- For **high performance**, use caching or optimized algorithms.
- For **high reliability**, include redundancy and fault tolerance.
- For **high security**, implement encryption and authentication layers.

Relationship to Architecture

Prioritizing quality features strongly influences **architectural decisions**:

- High **performance** → may suggest microservices or caching layers.
- High **security** → may suggest layered or zero-trust architectures.
- High **maintainability** → may favor modular or service-oriented designs.

Architecture is ultimately a reflection of the quality attributes the team values most.

Conclusion

Prioritizing quality features ensures that software not only *works* but also *works well* under real-world conditions.

It guides architectural design, technology selection, and testing strategies.

In essence, successful software is not just functional — it delivers the right **quality** in the areas that matter most to users and the business.

5.3. Selecting the Architectural Style or Approach in Software Engineering

In software engineering, **selecting the architectural style or approach** means **deciding on the overall structure and organization of a software system** — how its components are arranged, how they interact, and how responsibilities are divided.

An **architectural style** (also called an **architecture pattern**) provides a **set of design principles, rules, and conventions** for building systems with predictable qualities (like scalability, flexibility, or maintainability).
In short: Choosing an architectural style means deciding *how the system will be built and how its parts will work together.*

Choosing the right architectural style is a **strategic decision** that affects every stage of software development.
A well-chosen style ensures:

- **System reliability and performance**
- **Ease of maintenance and evolution**
- **Scalability and extensibility**
- **Clear separation of concerns**
- **Efficient collaboration among development teams**

A poor choice, on the other hand, can lead to:

- Complex dependencies
- Slow performance
- Difficulty in scaling or integrating new features
- High maintenance costs

Factors to Consider When Selecting an Architectural Style

When architects choose an approach, they evaluate several key factors:

Factor	Description
Functional Requirements	What the system needs to do (e.g., real-time updates, data processing, user interaction).
Non-Functional Requirements	Performance, scalability, reliability, maintainability, and security needs.
System Size and Complexity	Large, distributed systems may require different architectures than small applications.
Team Expertise	What technologies and styles the team understands best.
Deployment Environment	On-premises, cloud-based, or hybrid deployment considerations.
Integration Needs	Whether the system must connect to legacy systems, APIs, or external services.
Budget and Time Constraints	Some architectures take longer to implement or require more resources.

Common Architectural Styles in Software Engineering

Architectural Style	Description	Example Use Cases
Layered (n-Tier) Architecture	System divided into layers (Presentation, Business Logic, Data). Each layer communicates only with the one below it.	Traditional enterprise and web apps
Client-Server Architecture	A client requests services from a centralized server.	Web, email, or file server systems
Microservices Architecture	System divided into small, independent services that communicate via APIs.	Scalable, cloud-based systems
Event-Driven Architecture	Components communicate through asynchronous events or messages.	Real-time analytics, IoT, trading platforms
Service-Oriented Architecture (SOA)	Uses reusable services connected through an enterprise service bus (ESB).	Large organizations integrating multiple systems
Pipe-and-Filter Architecture	Data flows through a sequence of processing steps (filters).	Data processing, compiler design
Model-View-Controller (MVC)	Separates data (Model), presentation (View), and logic (Controller).	Web and desktop applications
Peer-to-Peer (P2P) Architecture	Each node acts as both a client and server.	File sharing, blockchain networks
Serverless / Cloud-Native Architecture	Code runs in cloud-managed services; scaling and infrastructure are automated.	Cloud applications, APIs, lightweight backends

Step-by-Step Process for Selecting an Architectural Style

1. **Analyze Requirements**
   Understand the system's goals, constraints, and environment.
2. **Identify Key Quality Attributes**
   For example: if *scalability* and *independent deployment* are priorities, **Microservices** may be best; if *simplicity* and *speed* matter, **Layered Architecture** may suffice.
3. **Consider Constraints**
   Assess available budget, time, and expertise.
4. **Compare Alternative Styles**
   Evaluate the pros and cons of multiple approaches using a decision matrix.
5. **Prototype and Validate**
   Build a small proof-of-concept (PoC) to test performance and integration.
6. **Document and Communicate the Decision**
   Record why the specific architecture was chosen and how it meets requirements.

Relationship with System Architecture

Selecting the architectural style defines the **foundation** upon which the system architecture is built.

It determines:

- Component structure
- Communication mechanisms
- Data flow
- Deployment model

In other words, the architectural style sets the **rules of the game** for how the system will evolve.

Conclusion

Selecting the right architectural style or approach is one of the **most critical decisions** in software design. It shapes the system's flexibility, scalability, and long-term success.

The best architectural style is the one that fits **your system's goals, context, and constraints** — not necessarily the most modern or complex one.

5.4. Defining Architectural Components in Software Engineering

In software engineering, **defining architectural components** means **identifying, describing, and organizing the main building blocks** of a software system. Each component represents a **distinct unit of functionality** that performs specific tasks within the architecture. In other words, components are **the "modules" or "parts" of the system's architecture** that work together to achieve the overall goals of the software.

Defining architectural components is a key step in **software architecture design**. It ensures that the system is:

- **Modular** (divided into manageable parts)
- **Understandable** (easy to explain and reason about)
- **Maintainable** (changes in one part have minimal impact on others)
- **Scalable** (new components can be added easily)
- **Reusable** (components can be used in other systems)

A well-defined architecture provides a *blueprint* for developers and helps coordinate large, complex projects efficiently.

What is an Architectural Component?

An **architectural component** is an **independent, replaceable unit** with a clear **responsibility, interface and behavior**.

Each component typically includes:

- **Responsibilities:** What it does (its purpose and functionality)
- **Interfaces:** How it communicates with other components
- **Dependencies:** What other components or services it relies on
- **Implementation Details:** Internal logic or technology stack (optional in high-level design)

Examples of Common Architectural Components:

Type of Component	Typical Role in System	Example Technologies
User Interface (UI)	Handles user interaction	React, Angular, Swift
Business Logic Layer	Implements rules and workflows	Java/Spring, C#/.NET, Node.js
Data Access Layer	Manages database operations	Hibernate, Entity Framework
Database / Storage	Stores persistent data	MySQL, MongoDB, PostgreSQL
API Gateway / Service Layer	Handles communication between clients and backend services	Express.js, FastAPI, Kong
Authentication Service	Manages login, roles, and security	OAuth2, Firebase Auth
Messaging / Integration Layer	Enables communication between distributed components	Kafka, RabbitMQ
Logging & Monitoring	Tracks system behavior and performance	Prometheus, ELK Stack

Steps to Define Architectural Components

When defining architectural components, engineers follow a structured process:

1. **Analyze System Requirements:** Understand what the system must do — both functional and non-functional requirements.
2. **Identify Major Responsibilities:** Group related features or operations into logical categories (e.g., user management, payment, reporting).
3. **Define Components and Their Boundaries:** Decide what each component does and what it doesn't do. Each should have a single, clear purpose.
4. **Specify Interfaces and Communication:** Define how components exchange data — via APIs, events, message queues, or shared databases.
5. **Establish Dependencies:** Determine which components depend on others and ensure loose coupling (minimal interdependence).
6. **Document the Components**
   Use diagrams, tables, and descriptions to record each component's role, interface, and interactions.

Relationship to Architectural Styles:

Different **architectural styles** influence how components are defined and connected:

- **Layered Architecture:** UI → Business Logic → Data Layer
- **Microservices Architecture:** Independent services communicating via APIs
- **Client-Server Architecture:** Clients request data from centralized servers
- **Event-Driven Architecture:** Components react to and emit events

The architectural style determines **how components interact and depend on each other**.

5.5. Types Of Architecture

5.6.1. Layered Architecture

Layered Architecture is one of the most common and foundational architectural styles in software engineering. It organizes the system into **vertical layers**, where each layer has a specific role and communicates only with its adjacent layers.

Layered Architecture (Also Called: n-Tier Architecture) divides a software system into layers, where each layer has a specific job and interacts only with the layer directly above or below it.
This creates a clear separation between:

- **Presentation** (UI)
- **Business logic**
- **Application workflow**
- **Data access**

Presentation Layer (UI Layer):

- What the user sees and interacts with
- Contains screens, views, controllers, web pages

Goal: Display information + send user requests to lower layers

Application Layer (Service / Application Logic)

- Coordinates the system's workflow
- Orchestrates business logic operations
- No business rules here—just process flow

Business / Domain Layer

- Contains **business rules**, **entities**, **validations**, **domain services**
- The **core logic** of the application
- Should be independent from UI or database

Data Access Layer (DAL) / Infrastructure Layer responsibles for database communication, file storage and external APIs. It includes repositories, ORMs, data mappers

Visualization:

```
------------------------
|  Presentation Layer  |
-------------------------
| Application Layer   |
-------------------------
|  Business Layer     |
-------------------------
| Data Access Layer   |
-------------------------
```

Key Ideas of Layered Architecture

- **Separation of concerns:** Each layer handles a different aspect of the system.
- **Loose coupling:** Layers depend on *abstractions*, not implementations.
- **Structured organization:** Easy for new developers to understand:
- "UI → Services → Business → Database"
- **Replaceability:** You can replace one layer without modifying others (e.g., switch database technology)

How Layers Communicate:

Allowed:

- UI → Application layer
- Application layer → Business layer
- Business layer → Data layer

Not allowed:

- UI accessing the database directly
- Data layer calling the UI
- Skipping layers (unless the architecture explicitly allows it)

Advantages of Layered Architecture:

- Very easy to understand
- Well-organized and maintainable
- Easy to test (mock layers)
- Stable, mature pattern used in enterprise software
- Good for medium to large applications

Disadvantages / Limitations:

- Can become slow due to strict layer-to-layer communication
- May lead to "anemic" architectures if business logic leaks into the application layer
- Hard to scale for large distributed systems
- Not ideal for high-performance microservices

Layered architecture organizes software into stacked layers, each responsible for a specific role, communicating only with adjacent layers to maintain clarity and separation of concerns.

Defining architectural components is one of the **most critical steps** in designing a robust and maintainable software system. By clearly identifying each component's **role, boundaries, and interactions**, engineers create a system that is easier to **understand, f**aster to **develop, s**impler to **scale and maintain.** Well-defined components turn complex systems into structured, manageable, and efficient architectures.

5.6.2. Microservices Architecture

Microservices architecture structures a system as a set of small, autonomous services. Each service runs independently, has its own database, and communicates with others through lightweight APIs.

Each microservice:

- focuses on one business function
- can be developed, deployed, and scaled independently
- is owned by a small team
- communicates via HTTP/REST, gRPC, message queues (Kafka, RabbitMQ)

High-Level Characteristics

1) Small and Autonomous

   Each microservice does *one thing well,* has its own logic, data storage, and lifecycle. Example services: payment service, user service, order service, ınventory service.

2) **Independent Deployment:** You can deploy one service without touching others. This enables continuous delivery and rapid updates.

3) **Decentralized Data: Each service owns its own database, preventing tight coupling. Example:**
   - User Service → users_db
   - Order Service → orders_db
   - Product Service → products_db

   Communication via APIs: Services talk to each other through REST APIs, gRPC, Messaging (Kafka, RabbitMQ). This keeps them loosely coupled.

Benefits of Microservices:

1. Scalability: Scale only the service that needs more resources (e.g., scale Order Service during Black Friday).
2. Technology Flexibility: Each microservice can be written in a different language/framework.
3. Resilience: If one service fails, the whole system doesn't crash.
4. Faster Development: Every teams work independently no bottlenecks.
5. Continuous Deployment: Deploy updates without downtime or affecting other services.

Limitations / Challenges:

- Operational Complexity: More moving parts → more monitoring, logging, load balancing.
- Distributed System Problems: Network latency (Gecikme), failures, retries, timeouts.
- Data Consistency Issues: Because databases are separate: Requires eventual consistency, Requires special patterns (Sagas, event sourcing),
- Harder Testing: Integration tests become more complex.

When to Use Microservices?

- Large applications
- Multiple independent teams
- Need for rapid updates & scaling
- High traffic systems (Uber, Netflix, Amazon)
- Complex domains requiring modularity

When Not to Use?

- Small teams
- Small or simple applications
- Early-stage startups
- Lack of DevOps / cloud infrastructure

Microservices architecture builds an application as a collection of small, independent, loosely-coupled services, each focused on a single business capability and deployable on its own.

Autonomous services:

Autonomous services is a network or software system. These systems can operate, make decisions, and manage itself with minimal human control.

Core characteristics of autonomous services:

- Self-configuring (kendi kurulumunu yapar)
- Self-healing (hataları algılar ve düzeltir)
- Self-optimizing (performansı kendi iyileştirir)
- Self-protecting (kendini tehditlere karşı korur)

Where it is used?
Modern networks use this idea in autonomic networking, inspired by the scientific concept of Autonomic Nervous System (the system that controls breathing, heart rate, etc., automatically). Cloud platforms provide autonomous service capabilities, such as:

- Amazon Web Services
- Microsoft Azure

AI-driven autonomous services examples:

- Self-managing IT monitoring by Datadog
- Automated incident management by PagerDuty

Everyday analogy, think of a service that works like:

- A self-driving car (acts without constant driver input)
- A smart thermostat that adjusts temperature on its own
- A network system that detects congestion and reroutes traffic automatically

Short definition

- Autonomous services = services that run and manage themselves intelligently.
- They detect situations, decide what to do, and act automatically (using rules, automation, or AI).

5.6.3. Client–Server Architecture

Client–Server Architecture divides a system into two major components:

- a Client requests and consumes services
- a Server provides and manages services

These two communicate over a network (LAN, WAN, internet).

The client does the front-end tasks (UI, sending requests), and the server does the back-end tasks (processing, storage, logic).

A *client* is typically a web browser, a mobile app, a desktop application. The client is responsible for displaying UI, capturing user input, sending requests to the server, presenting the server's response, Clients are usually thin (minimal logic) or thick (fat) (more logic).

A server is a computer or software system that provides resources, services, or data to other computers, called clients, over a network. According to this sentence, what is the client? Clients communicate each other or servers. A client is a device (computer) or program that requests a service or data from another system (server or client).

A *server* is a powerful machine or service responsible for processing client requests, executing business logic. The server accesses and manages databases, for returning responses. The server is the "brains" of the system.

A host is any computer, device, or system that is connected to a network and can send or receive data. It has a unique address (like an IP address) so other devices can find it. It can act as a client, a server, or both depending on the situation. On the internet, websites live on web hosts—servers that keep content available online.

Common host examples:

- Your laptop or phone is a host when it connects to Wi-Fi.
- Apache and Nginx are software that turn a device into a server-type host.
  So think of it like this:
- Host = network üzerindeki bir katılımcı (iletişim kurabilen cihaz)
- Client = istek yapan host
- Server = hizmet sunan host

Typical server examples:

- Web servers (Apache, Nginx)
- Application servers (.NET, Java Spring)
- Database servers (MySQL, PostgreSQL)

How Client–Server Architecture works (Flow)

1. Client sends a request
2. Server receives it
3. Server processes it (logic + database)
4. Server sends back a response
5. Client displays the result

Advantages of Client–Server Architecture:

- Centralized control: Logic and data live on the server; easy to update and secure.
- Easier maintenance: Changing the server changes behavior for all clients.
- Scalable: Servers can be upgraded or replicated.
- Security: Server controls permissions and sensitive data.

Disadvantages of Client–Server Architecture

- Server dependency: If the server fails, the system is unavailable.
- Network dependency: Requires network connectivity.
- Bottlenecks: Too many clients, so high load on the server.

Common Real-World Examples of Client–Server Architecture:

- Web applications: Browsers (clients) → Web server → Database
- Mobile apps: Mobile app (client) → REST API server → Database
- Online games: Game client → Game server
- Email: Email client → Email server

Types of Client–Server Models

1. Two-tier architecture

- Client ↔ Server
- Example: simple database applications

2. Three-tier architecture

- Client
- Application server
- Database server
- Common in enterprise systems

3. Multi-tier architecture

- More layers for caching, load balancing, microservices, etc.

Client–Server Architecture separates a system into clients that request services and servers that deliver, process, and manage those services over a network.

5.6.4. Event-Driven Architecture (EDA)

Event-Driven Architecture is a design approach where system components interact by generating and responding to events. Components of EDA do *not* call each other directly; instead, they communicate through events broadcast via an event broker or message system.

An event is a significant change in state. Examples:

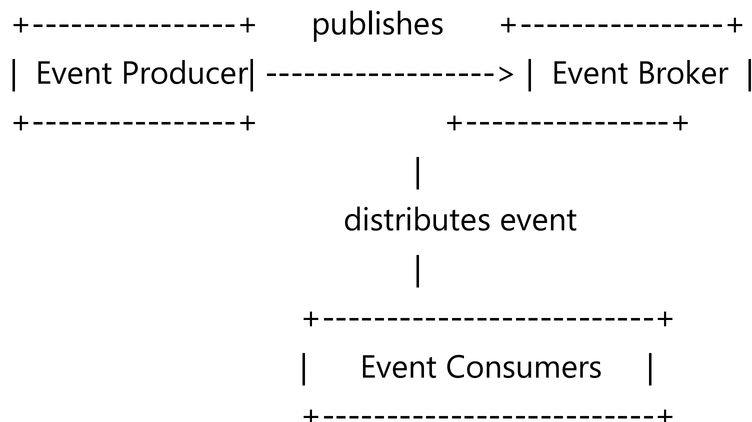
- "OrderPlaced"
- "PaymentCompleted"
- "UserRegistered"
- "InventoryLow"
- "FileUploaded"

Each event is broadcast and any component interested in that event can react.

Core Components of EDA:

- Event Producer: Creates and publishes events. Example: Order Service publishes "OrderPlaced".
- Event Consumer: Listens for and reacts to events. Example: Notification Service listens to "UserRegistered".
- Event Broker / Event Bus: A middleware that routes events. Examples:
  - Kafka
  - RabbitMQ
  - AWS SNS/SQS
  - Azure Event Hub

Architecture Diagram (Simplified)



Why Use Event-Driven Architecture?

- Loose Coupling: Producers don't know anything about consumers.
- High Scalability: Events can be processed by many consumers in parallel.
- Asynchronous Processing: No waiting. Producers publish events and move on.
- Real-Time Reactions: Used for systems that need instant updates.
- Flexibility: New consumers can be added without modifying producers.

Example Scenario: E-Commerce

Event: "OrderPlaced"

1. Order Service publishes OrderPlaced
2. Different consumers react:
    - Payment Service → processes payment
    - Inventory Service → updates stock
    - Notification Service → sends email
    - Analytics Service → logs the event

They all work independently.

Challenges and Tradeoffs:

- Complexity: Distributed, asynchronous systems are harder to debug.
- Eventual Consistency: Data may not be updated everywhere at the same time.
- Monitoring Difficulty: Harder to track the flow of events across services.
- Requires Specialized Infrastructure: Kafka, RabbitMQ, streaming pipelines, etc.

Where EDA Is Used

- E-commerce platforms
- Financial trading systems
- Real-time analytics
- IoT systems
- Microservices communication
- Modern cloud-native applications

Event-Driven Architecture is a system design where components communicate by producing and reacting to events, enabling loose coupling, high scalability, and real-time responsiveness.

5.7. Selecting Technologies and Tools in Software Engineering

Selecting technologies and tools means **choosing the appropriate programming languages, frameworks, libraries, databases, development environments, and platforms** that will be used to design, build, test, deploy, and maintain a software system.

The right choice of technologies directly affects:

- **Development speed and cost**
- **System performance and scalability**
- **Maintainability and security**
- **Team productivity and satisfaction**
- **Long-term sustainability** of the project

Making poor technology choices early in a project can lead to **technical debt**, **integration problems**, and **increased maintenance costs** later on.

2. Key Considerations When Selecting Technologies

When deciding which tools and technologies to use, software engineers typically evaluate the following criteria:

Factor	Description
Project Requirements	Does the technology meet the functional and non-functional requirements (performance, reliability, scalability, etc.)?
Team Expertise	Do developers already have experience with the technology? If not, how steep is the learning curve?
Community Support & Documentation	Is there a strong community, good documentation, and active maintenance?
Compatibility and Integration	*Can it integrate easily with other tools, systems, and APIs already in use?*
Security	Does the technology have known vulnerabilities or strong security practices?
Cost	Is it open-source, licensed, or subscription-based? What is the total cost of ownership (TCO)?
Performance and Scalability	Can it handle the expected load and grow with future needs?
Maintainability	Will it be easy to update, debug, and modify in the future?
Longevity and Stability	Is the technology widely adopted and likely to remain supported for years?

3. Common Categories of Technologies and Tools

1. **Programming Languages:**
   Examples: Python, Java, C#, JavaScript, Go, Rust
   → Chosen based on performance needs, platform targets, and developer familiarity.
2. **Frameworks and Libraries:**
   Examples: React, Angular, Spring Boot, Django, .NET
   Provide reusable code and structure to speed up development.
3. **Databases:**
   Examples: MySQL, PostgreSQL, MongoDB, Redis
   → Chosen based on data structure (relational vs. NoSQL), scalability, and transaction needs.
4. **Version Control Systems:**
   Example: Git (GitHub, GitLab, Bitbucket)
   → Essential for collaborative development and change tracking.
5. **CI/CD and DevOps Tools:**
   Examples: Jenkins, Docker, Kubernetes, GitHub Actions
   → Help automate testing, deployment, and scaling.
6. **Testing and Quality Assurance Tools:**
   Examples: JUnit, Selenium, Postman
   → Ensure code reliability and performance.
7. **Project Management Tools:**
   Examples: Jira, Trello, Asana
   → Facilitate communication, planning, and task tracking.

4. Decision-Making Process

A structured process is often followed:

1. **Requirement Analysis** – Understand technical and business needs.
2. **Market Research** – Identify available technologies and alternatives.
3. **Evaluation and Comparison** – Create a comparison matrix (pros/cons, features, support).
4. **Prototyping or Proof of Concept (PoC)** – Test technologies on a small scale.
5. **Final Selection** – Choose based on results, team input, and long-term vision.

5. Example Scenario

A startup is developing a **web application for real-time chat**:

- Requirements: scalability, low latency, cross-platform support.
- Decision:
  - Backend: **Node.js** (handles real-time communication efficiently)
  - Frontend: **React.js**
  - Database: **MongoDB** (NoSQL, flexible schema)

- Deployment: **Docker + AWS ECS**

This combination supports the project's scalability and team expertise while minimizing setup time.

6. Conclusion

Selecting technologies and tools is **a strategic decision** in software engineering that combines **technical judgment, business understanding, and team dynamics**.
The goal is not just to pick the "latest" or "most popular" technology, but the **most suitable and sustainable** one for the specific project context.

5.7.1. Programming Languages

In software engineering, programming languages are structured systems of rules and syntax used to write instructions that computers, servers, or other digital systems can understand and execute.

Programming Languages allow software engineers to:

- build applications (web, mobile, desktop)
- communicate with hardware or system components
- process and store data
- automate tasks
- create intelligent or autonomous behavior

Main aspects of programming languages

1. Syntax (grammar rules)

Defines how code must be written.
Example: In Python you print text with print("Hello"), In Java you write System.out.println("Hello");

2. Semantics (meaning of instructions)

Defines what each statement actually *does* when executed.

3. Abstraction level

Level	Meaning	Examples
High-level	Easy for humans, far from hardware details	Python, Java, JavaScript
Low-level	Closer to machine, controls memory and hardware	C, Assembly (generic low-level language)

4. Language paradigms (style of thinking)

Describes how problems are modeled:

- **Object-Oriented (OOP)** → models with objects: Python, Java, C++
- **Functional** → models with math-like functions: Haskell
- **Procedural** → step-by-step instructions: C
- **Scripting** → automation and glue for systems: Python, Bash (linux scripting shell)

5. Compilation or interpretation

Method	Meaning	Example languages
Compiled	Converted to machine code before running	C, C++
Interpreted	Runs instruction by instruction	Python, JavaScript

Why they matter

Programming languages are essential because:

✅ They bridge human logic and machine execution
✅ They enforce clarity, correctness, and maintainability
✅ They enable large-scale systems and collaborative development

A programming language is a tool that lets software engineers express **algorithms**, handle **data**, and control **systems** so that digital devices perform useful work, reliably and efficiently.

5.7.2. Framework

In software engineering, a framework is a reusable foundation (a set of tools, libraries, and rules) that helps engineers build software faster, with less code, and in an organized way.

A framework usually provides:

- a **project structure**
- **ready-made components** (buttons, forms, database access, authentication, routing, etc.)
- **best practices**
- rules about **how your code should interact** with the system

Unlike a normal library (which you call when you want), a framework often **calls your code** and controls the flow. This principle is known as **Inversion of Control (IoC)**.

Examples by domain:
Web development Frontend: React, Angular; Backend: Django, Express.js (Node.js backend framework)
Mobile app development: Flutter, React Native
Game development: Unity, Unreal Engine

Key benefits of framework:
✅ **Speeds up development** (no need to reinvent common features)
✅ **Improves maintainability** (standard structure)
✅ **Encourages teamwork** (everyone codes in the same environment and logic)
✅ **More secure and optimized** (solutions tested by large communities)

Simple analogy:
Think of a framework like:

Construction analogy	Software meaning
A pre-built house skeleton	The framework sets up the structure
Electric and water systems already installed	Ready-made modules & tools
Rules for building safely	Coding standards & architecture

You only focus on customizing the inside — not building the whole house from scratch.

A framework is a **software toolbox + blueprint** that guides how to develop an application while handling most of the heavy technical groundwork for you.

5.7.3. Libraries

In software engineering, a library is a collection of pre-written code that performs specific tasks, so engineers can reuse functionality without writing everything from scratch.

You **call the library** when you need it — it does *not* control your whole app like a framework does.

What libraries provide, Libraries help you:

- work with data, math, networking, files, images, APIs…
- add features like encryption, HTTP requests, UI widgets, etc.

Libraries typically include reusable:

- functions
- classes
- methods
- modules

Examples of popular libraries (by purpose):
Mathematics & data processing

- NumPy – advanced numerical computing
- Pandas – data manipulation and analysis

HTTP / networking

- Requests – simple HTTP requests in Python
- Axios – promise-based HTTP client for JS

Machine Learning

- TensorFlow – ML and deep learning library
- Scikit-learn – classical ML algorithms

UI / visualization

- Matplotlib – plotting and charts in Python
- D3.js – interactive visualizations for the web

Library vs Framework (quick distinction)

Library	Framework
You control the flow	It controls the flow
You call it	It may call your code
Used for specific needs	Provides whole project foundation
Smaller scope	Larger scope

A library is like going to a store and buying a ready-made tool (hammer, drill, paint). You decide **when and how to use it**. Libraries are **reusable code tools** that give you solutions for **specific problems**, helping you write clean, shorter, and more efficient software — while keeping control fully in your hands.

5.7.4. Databases

In software engineering, **databases** are systems that **store, organize, and manage data** so that software can **retrieve, update, and analyze** it efficiently. Think of them as a **digital memory** for applications—structured, searchable, and persistent.

Core characteristics

- **Store data permanently** (even if the app restarts)
- Allow **fast searching and filtering**
- Support **multiple users/processes safely**
- Maintain **data consistency and security**
- Provide rules for data structure (schema, relations, indexing, etc.)

Main types

1) Relational Databases (SQL)

Stores data in **tables (rows & columns)** with defined relationships.

- Examples: PostgreSQL, MySQL, Microsoft SQL Server
- Uses query language: **SQL (Structured Query Language)**

2) NoSQL Databases

More flexible than tables; can store **documents, key-value, graphs, or wide columns**

- Examples: MongoDB, Redis, Neo4j

Key concepts you'll hear

- **Schema**: structure/format of data
- **Query**: asking the database for data
- **Index**: helps search faster
- **Transactions**: ensures safe, reliable updates
- **Backup & replication**: protects against data loss
- **ACID**: rules for data reliability (common in relational DBs)

Simple analogy

- **Database** = the library
- **Tables/Documents** = bookshelves or books
- **SQL/Query** = asking the librarian
- **Index** = the catalog that speeds up search

Databases are structured storage engines that let software work with data reliably and at scale.

5.7.5. Development Environments

In software engineering, **development environments** are the **tools + setup** that developers use to **build, test, and run software**. They include editors, compilers, databases, servers, configuration files, and automation tools—everything needed to develop efficiently.

Common environment types:

- **Local Development Environment**
  Runs on the developer's computer. Uses tools like Visual Studio Code, languages installed via Node.js, containers with Docker.
- **Integrated Development Environment (IDE)**
  All-in-one software for coding, debugging, and compiling. Examples: IntelliJ IDEA, PyCharm.
- **Virtual/Containerized Environment**
  Isolated systems that behave the same everywhere. Managed with Docker Compose or cloud sandboxes.
- **Cloud Development Environment**
  Runs in the browser, hosted remotely. Examples: GitHub Codespaces or similar platforms.

Core components usually included

- Code editor / IDE
- Compiler or interpreter
- Debugging tools
- Version control (like Git)
- Runtime environment
- Databases or API services
- Automation & build tools
- Testing frameworks
- Environment variables & configuration

Why they matter

- Make coding faster
- Reduce errors
- Keep systems **consistent, secure, and reproducible**
- Support testing before release
- Simulate real production systems safely

A development environment is the workspace where software is engineered and validated before deployment.

5.7.6. Platforms

In software engineering, a **platform** is a **foundation or environment** on which **applications or software systems run**. It provides the **tools, frameworks, and services** needed for development, deployment, and execution.
Think of it as the **stage where software performs**—without it, the software can't function properly.

Key aspects of platforms:

- **Runtime environment**: Executes applications (e.g., operating systems, virtual machines)
- **APIs & SDKs**: Provide pre-built functions to interact with hardware, services, or other software
- **Services & tools**: Facilitate development, deployment, or maintenance
- **Hardware/OS support**: Ensures compatibility with underlying infrastructure

Common types of platforms:

1. **Operating System Platforms**
    - Provide core services for apps (file system, memory management, networking)
    - Examples: Windows, Linux, macOS, Android, iOS
2. **Cloud Platforms**
    - Offer computing, storage, and services over the internet
    - Examples: AWS, Microsoft Azure, Google Cloud Platform
3. **Application Platforms**
    - Provide environment for specific software types
    - Examples: WordPress (website), Salesforce (CRM), Unity (game development)
4. **Hardware Platforms**
    - Specific hardware with supporting software
    - Examples: Raspberry Pi, PlayStation, Arduino

Simple analogy

- **Platform = stage**
- **Application = actor**
- **APIs & services = props and crew**
- Without the stage, the actor cannot perform properly.

Platforms are the environments and foundations that software relies on to run, interact with other software/hardware, and provide functionality.

5.8. Preparing Architectural Diagrams in Software Engineering

In software engineering, **architectural diagrams** are **visual representations of a system's structure**. They show how **components, modules, services, databases, and external systems** interact to deliver the system's functionality. The goal is to **communicate the high-level design** of the software clearly — both to technical and non-technical stakeholders.

Architectural diagrams help:

- Visualize **system organization**
- Identify **dependencies** and **interfaces**
- Support **decision-making** in design and development
- Simplify **communication** within the development team

  In short, architectural diagrams act as a *blueprint* for the software system.

Importance of Architectural Diagrams

Creating architectural diagrams is essential for:

- **Documentation:** Provides a long-term reference for developers and maintainers
- **Planning:** Helps in resource allocation and workload distribution
- **Collaboration:** Ensures all team members share the same understanding of the system
- **Troubleshooting:** Makes it easier to locate faults or bottlenecks in the system

Key Principles When Preparing Diagrams

When preparing architectural diagrams, engineers should focus on:

Principle	Description
Clarity	Avoid unnecessary details; show only what's relevant at the current level.
Consistency	Use consistent symbols, colors, and notation styles.
Simplicity	Each diagram should serve one purpose (don't mix too many views).
Abstraction Levels	Different diagrams represent different levels — from high-level overview to low-level design.
Accuracy	Ensure all connections, data flows, and relationships are correct.

Common Types of Architectural Diagrams

Type	Purpose	Typical Elements
System Context Diagram	Shows how the system interacts with users and external systems.	Actors, external systems, input/output flows
Component Diagram	Displays internal structure — components, modules, and their interactions.	Modules, interfaces, dependencies
Deployment Diagram	Describes how software components are deployed on hardware or cloud infrastructure.	Servers, containers, networks, databases
Data Flow Diagram (DFD)	Illustrates how data moves through the system.	Processes, data stores, inputs/outputs
Sequence Diagram	Shows the order of interactions between system components over time.	Messages, lifelines, sequence of events
Microservices or Cloud Architecture Diagram	Visualizes distributed services, APIs, and their communication.	Services, gateways, load balancers, message queues

Common Tools for Drawing Architectural Diagrams

Software engineers often use diagramming tools to create clean and professional visuals, such as:

- **Draw.io (diagrams.net)**
- **Lucidchart**
- **Microsoft Visio**
- **PlantUML** (for text-based diagrams)
- **Cacoo, Figma, or Miro** (for collaborative design)
- **AWS Architecture Icons** or **Azure Diagrams** for cloud architectures

Step-by-Step Process

1. **Define the Purpose**
   Decide what you want to communicate — system overview, data flow, deployment layout, etc.
2. **Identify Key Components**
   List main modules, services, databases, APIs, and external systems.
3. **Determine Relationships**
   Define how components communicate — via REST APIs, message queues, databases, etc.
4. **Choose Diagram Type and Notation**
   Select UML, C4 model, or another notation appropriate for your audience.
5. **Draw and Label Clearly**
   Use standard symbols, add arrows for data flow, and include legends if necessary.
6. **Review and Validate**
   Share with team members to confirm accuracy and completeness.

Example Scenario

Imagine a **web-based e-commerce system**:

- **Frontend:** React web app
- **Backend:** Node.js microservices
- **Database:** PostgreSQL
- **Payment Gateway:** External service
- **Deployment:** Docker containers on AWS

The **architectural diagram** would show:

- User interacting with the web app
- Web app communicating with backend services
- Backend accessing the database
- Integration with the payment API
- All components hosted on cloud infrastructure

This diagram would help the team understand the system flow and deployment at a glance.

Conclusion

Preparing architectural diagrams is a **fundamental skill** in software engineering.
They serve as a **bridge between abstract ideas and concrete implementation**, ensuring that everyone—from developers to managers—shares the same mental model of the system.
Well-designed diagrams make complex architectures **understandable, maintainable, and scalable**.

6. Preparing diagrams and presentations in software architecture

Preparing diagrams and presentations in software architecture is a critical step for accurately describing a system's design, establishing a shared understanding with stakeholders, and documenting decisions. Let me provide comprehensive step-by-step information:

1. The Importance of Diagrams in Software Architecture

Simplifies complex structures → Makes complex systems more understandable by visualizing them.

Communication tool → Provides a common language with developers, business analysts, managers, and even customers.

Decision record → Allows documentation of which technologies, structures, and integrations were selected.

Predicts errors → Provides early detection of future scalability, security, and performance problems.

2. Types of Diagrams Used in Software Architecture

Diagrams are necessary at different levels when describing architecture. Most commonly used:

(A) Conceptual Level

Context Diagram

Shows the system's relationships with the outside world.

Who does it interact with (users, other systems)?

Business Capability Diagram

Shows which software components support business processes.

(B) Logical Level

Component Diagram

Shows the system's decomposition into modules (e.g., User Management, Payment System, Notification Service).

Class/Package Diagram

Classes and their relationships in an object-oriented approach.

(C) Physical/Technical Level

Deployment Diagram

Servers, databases, APIs, services, network structure.

Infrastructure Diagram

Cloud services (AWS, Azure, GCP), load balancers, cache, queue systems.

(D) Dynamic Level
Sequence Diagram

For example: "What happens when the user logs in?" Shows step-by-step.
Activity Diagram / Flowchart
Shows workflows.

(E) C4 Model (a highly recommended approach)
Simon Brown's recommended 4-level model:
Context → How does the system communicate with the outside world?
Container → Main modules, services, databases.
Component → Modules within services.
Code → Optional, at the class/object level.

3. Considerations When Creating Diagrams
Simplicity → Dividing into levels instead of too much detail.
Standard notation → One of the standards such as UML, ArchiMate, or BPMN.
Consistency → The same icon set, colors, and symbols across all diagrams.
Target Audience →
Management → Conceptual, business process-focused diagrams.
Developers → Technically detailed diagrams.
Customers → Simple, user journey-focused diagrams.

4. How to Prepare a Presentation
In a software architecture presentation:
Start: The Big Picture:

- What does the system solve?
- Key objectives (performance, scalability, security, etc.)

Conceptual Architecture

- Show the system's relationship to the outside world with a context diagram.

Logical Architecture

- Explain how modules/components are separated.

Technical Architecture

- Show database, API, microservice, and cloud architectures.

Dynamic Flows

- Include sequence diagrams for key user scenarios.

Non-Functional Requirements (NFR)

- Performance, security, usability, maintainability.

Decisions and Alternatives

- Which architectural decisions were made and why?

- What were the alternative solutions?

Conclusion and Roadmap

- Benefits of the architectural design for the company/product.
- Future developments.

5. Available Tools

- Draw.io / diagrams.net → Free, simple, and widespread.
- Lucidchart → Easy collaboration.
- PlantUML / Mermaid → Code-based diagramming.
- ArchiMate Tools (Archi, BizzDesign) → For enterprise architecture.
- Whimsical, Miro → Brainstorming and collaborative diagramming.

Yazılım mimarisinde diyagram ve sunum hazırlama

Yazılım mimarisinde **diyagram ve sunum hazırlama**, bir sistemin tasarımını doğru şekilde anlatmak, paydaşlarla ortak bir anlayış oluşturmak ve kararları belgelendirmek için çok kritik bir adımdır. Sana adım adım kapsamlı bilgi vereyim:

1. Yazılım Mimarisinde Diyagramların Önemi

- **Karmaşık yapıyı basitleştirir** → Karmaşık sistemleri görselleştirerek daha anlaşılır hale getirir.
- **İletişim aracı** → Geliştiriciler, iş analistleri, yöneticiler, hatta müşteri ile ortak bir dil sağlar.
- **Karar kaydı** → Hangi teknolojilerin, hangi yapıların ve hangi entegrasyonların seçildiğini belgeleme imkanı verir.
- **Hataları önceden görme** → İleride yaşanabilecek ölçeklenebilirlik, güvenlik, performans problemlerini erken fark etmeyi sağlar.

2. Yazılım Mimarisinde Kullanılan Diyagram Türleri

Mimariyi anlatırken farklı seviyelerde diyagramlar gerekir. En çok kullanılanlar:

(A) Kavramsal Seviye

- **Context Diagram (Bağlam Diyagramı)**
  - Sistemin dış dünya ile ilişkilerini gösterir.
  - Kimler (kullanıcılar, diğer sistemler) ile etkileşime giriyor?
- **Business Capability Diagram**
  - İş süreçlerinin hangi yazılım bileşenleri tarafından desteklendiğini gösterir.

(B) Mantıksal Seviye

- **Component Diagram (Bileşen Diyagramı)**
  - Sistemin modüllere ayrılışını gösterir (örneğin: Kullanıcı Yönetimi, Ödeme Sistemi, Bildirim Servisi).

- **Class / Package Diagram**
  - Nesne tabanlı yaklaşımda sınıflar ve ilişkileri.

(C) Fiziksel / Teknik Seviye

- **Deployment Diagram (Dağıtım Diyagramı)**
  - Sunucular, veri tabanları, API’ler, servisler, ağ yapısı.
- **Infrastructure Diagram**
  - Bulut servisleri (AWS, Azure, GCP), yük dengeleyiciler, cache, queue sistemleri.

(D) Dinamik Seviye

- **Sequence Diagram (Sıra Diyagramı)**
  - Örneğin: “Kullanıcı giriş yaparken neler olur?” adım adım gösterir.
- **Activity Diagram / Flowchart**
  - İş akışlarını gösterir.

(E) C4 Model (çok önerilen bir yaklaşım)

Simon Brown’ın önerdiği 4 seviyeli model:

1. **Context** → Sistem dış dünya ile nasıl iletişimde?
2. **Container** → Ana modüller, servisler, veri tabanları.
3. **Component** → Servislerin içindeki modüller.
4. **Code** → İsteğe bağlı, sınıf/nesne seviyesinde.

3. Diyagram Hazırlarken Dikkat Edilecekler

- **Basitlik** → Çok fazla detay yerine seviyelere bölmek.
- **Standart notasyon** → UML, ArchiMate, BPMN gibi standartlardan biri.
- **Tutarlılık** → Tüm diyagramlarda aynı ikon seti, renk, sembol.
- **Hedef kitle** →
  - Yönetime → Kavramsal, iş süreçleri odaklı diyagramlar.
  - Geliştiricilere → Teknik detaylı diyagramlar.
  - Müşteriye → Sade, kullanıcı yolculuğu odaklı diyagramlar.

4. Sunum Hazırlarken İzlenecek Yol

Bir yazılım mimarisi sunumunda:

1. **Başlangıç: Büyük Resim**
    - Sistem neyi çözüyor?
    - Temel hedefler (performans, ölçeklenebilirlik, güvenlik vs.)
2. **Kavramsal Mimari**
    - Bağlam diyagramı ile sistemin dış dünya ile ilişkisini göster.
3. **Mantıksal Mimari**
    - Modüllerin/bileşenlerin nasıl ayrıldığını anlat.
4. **Teknik Mimari**
    - Veritabanı, API, microservice, cloud yapılarını göster.
5. **Dinamik Akışlar**
    - Önemli kullanıcı senaryoları için sequence diagram ekle.
6. **Non-Functional Requirements (NFR)**
    - Performans, güvenlik, kullanılabilirlik, bakım kolaylığı.
7. **Kararlar ve Alternatifler**
    - Hangi mimari kararlar neden alındı?
    - Alternatif çözümler nelerdi?
8. **Sonuç ve Yol Haritası**
    - Mimari tasarımın şirket/ürün için getirdiği faydalar.
    - Gelecek geliştirmeler.

5. Kullanılabilecek Araçlar

- **Draw.io / diagrams.net** → Ücretsiz, basit, yaygın.
- **Lucidchart** → Kolay işbirliği.
- **PlantUML / Mermaid** → Kod tabanlı diyagram.
- **ArchiMate Tools (Archi, BizzDesign)** → Kurumsal mimari için.
- **Whimsical, Miro** → Beyin fırtınası ve işbirlikçi diyagramlar.

7. Forming a software team to develop a project or solve a problem.

Bir proje hazırlamak ya da bir problem çözmek için yazılım ekibi oluşturulması.
Nasıl bir yöntem ve metotoloji izlemeliyim?
Nasıl planlama yapmalıyım?

Mimari yapı oluştururken nelere dikkat etmeliyim?
Bunlar, profesyonel yazılım geliştirme ve proje yönetiminin **çekirdeğini** oluşturur. Bir yazılım ekibi kurup proje yürütürken, **doğru metodoloji, iyi planlama** ve **sağlam mimari yapı** başarının üç temel direğidir. Aşağıda, adım adım kapsamlı bir yol haritası verilmiştir:

What methodology should we follow?
How should we plan?
What should we consider when creating an architectural structure?
These are the core elements of professional software development and project management. When building a software team and executing a project, the right methodology, good planning, and a solid architectural structure are the three pillars of success. Below is a comprehensive, step-by-step roadmap:

1. Stratejik Başlangıç: Amaç, Vizyon ve Kapsam
Amaç net değilse, hiçbir metodoloji işe yaramaz.
Yapılacaklar:

- **Projenin amacını** ve çözmek istediğin **problemi** tanımla. "Neyi, kimin için, neden yapıyoruz?"
- **Kapsamı belirle:** Ne dahil, ne hariç?
- **Başarı kriterleri** belirle: Zaman, kalite, maliyet, kullanıcı memnuniyeti gibi.

Araçlar:
Vision Document, Project Charter, Product Requirement Document (PRD)

1. Strategic Start: Purpose, Vision, and Scope
If the purpose isn't clear, no methodology will work.
Topics:
• Define the project's purpose and the problem you want to solve. "What are we doing, for whom, and why?"
• Define the scope: What's included and what's excluded?
• Establish success criteria: Time, quality, cost, user satisfaction, etc.
Tools: Vision Document, Project Charter, Product Requirement Document (PRD)

2. Ekip Yapısı ve Roller

Doğru ekip → Doğru ürün. Doğru ekip oluşturulmadan hiçbir şey üretemezsiniz.

Temel Roller:

- **Product Owner (Ürün Sahibi):** İhtiyaçları belirler, önceliklendirir. Sıralar, fazlandırır.
- **Project Manager / Scrum Master:** Planlama, koordinasyon, süreç yönetimi.
- **Tech Lead / Software Architect:** Teknik yön, mimari kararlar.
- **Developers (Frontend, Backend, Mobile, AI, vb.)**
- **QA / Tester:** Kalite kontrol.
- **UI/UX Designer:** Kullanıcı deneyimi.

Küçük projelerde roller birleştirilebilir (örneğin Tech Lead aynı zamanda Developer olabilir).

2. Team Structure and Roles

The right team → The right product. You can't produce anything without the right team.

Basic Roles:

• Product Owner: Identifies and prioritizes requirements. Sequences and phases.
• Project Manager / Scrum Master: Planning, coordination, and process management.
• Tech Lead / Software Architect: Technical direction, architectural decisions.
• Developers (Frontend, Backend, Mobile, AI, etc.)
• QA / Tester: Quality control.
• UI/UX Designer: User experience.

Roles can be combined on smaller projects (for example, the Tech Lead can also be a Developer).

3. Metodoloji Seçimi: Agile mı, Waterfall mu?

Agile (Önerilen)

- Esnek, yinelemeli (iteratif) yaklaşım.
- Gereksinimler değişebilir.
- Her sprint sonunda çalışan bir ürün parçası ortaya çıkar.

Alt yaklaşımlar:

- **Scrum:** Sprint'ler (2-4 hafta), backlog, daily stand-up.
- **Kanban:** Sürekli akış, görsel takip (Trello, Jira board).
- **Lean:** Minimum israf, sürekli iyileştirme.

Waterfall

- Net, değişmeyen gereksinimler için uygundur.
- Aşamalar sıralıdır: Analiz → Tasarım → Geliştirme → Test → Yayın.

3. Methodology Choice: Agile or Waterfall?
Agile (Recommended)
• Flexible, iterative approach.
• Requirements may change.
• A working product increment is released at the end of each sprint.

Sub-approaches:
• Scrum: Sprints (2-4 weeks), backlog, daily stand-up.
• Kanban: Continuous flow, visual tracking (Trello, Jira board).
• Lean: Minimal waste, continuous improvement.

Waterfall
• Suitable for clear, unchanging requirements.
• Phases are sequential: Analysis → Design → Development → Test → Release.

4. Planlama ve Yol Haritası
Plan olmadan hareket etmek, rotasız gemi gibidir.
Adımlar:
1. **Gereksinim Analizi (Requirements Analysis)**: Kullanıcı hikayeleri (user stories) ve kabul kriterleri.
2. **Zaman Planı (Timeline)**: Sprint bazlı planlama, milestone'lar.
3. **Kaynak Planı (Resource Planning)**: Kimin ne yapacağı, hangi araçlarla yapılacağı.
4. **Risk Analizi**: Teknik, finansal, operasyonel riskler.

Araçlar: Jira, ClickUp, Notion, Miro, Gantt Chart, Roadmap Planner

4. Planning and Roadmap
Operating without a plan is like a ship without a rudder.
Steps:
1. Requirements Analysis: User stories and acceptance criteria.
2. Timeline: Sprint-based planning, milestones.
3. Resource Planning: Who will do what, and with what tools.
4. Risk Analysis: Technical, financial, and operational risks.

Tools: Jira, ClickUp, Notion, Miro, Gantt Chart, Roadmap Planner

5. Mimari Yapı Oluşturma

Mimari yapı, projenin omurgasıdır.

5.1. Mimari İlkeleri

- **Modülerlik:** Her bileşen bağımsız geliştirilebilir.
- **Genişletilebilirlik (Scalability):** İleride artacak kullanıcı/iş yüküne hazır olmalı.
- **Güvenlik:** Veriler ve servisler korunmalı.
- **Test Edilebilirlik:** Kod kolay test edilebilmeli.
- **Sürdürülebilirlik:** Kod okunabilir, dokümante olmalı.

5.2. Mimarinin Türü (projeye göre)

- **Monolitik:** Küçük, basit projelerde.
- **Katmanlı (Layered Architecture):** Veri – İş Mantığı – Sunum katmanları.
- **Mikroservis:** Büyük, ölçeklenebilir sistemlerde.
- **Event-driven / Serverless:** Modern bulut tabanlı sistemlerde.

5.3. Teknik Seçimler

- **Frontend:** React, Vue, Angular
- **Backend:** Node.js, Python (Django/FastAPI), Java (Spring), .NET Core
- **Database:** PostgreSQL, MongoDB, Redis
- **CI/CD:** GitHub Actions, Jenkins
- **Cloud:** AWS, Azure, GCP
- **Container:** Docker, Kubernetes

5. Creating the Architectural Structure

The architectural structure is the backbone of the project.

5.1. Architectural Principles

• Modularity: Each component can be developed independently.

• Scalability: It should be ready for future user/workload increases.

• Security: Data and services must be protected.

• Testability: The code must be easy to test.

• Maintainability: The code must be readable and documented.

5.2. Architecture Type (by project)

• Monolithic: For small, simple projects.

• Layered Architecture: Data – Business Logic – Presentation layers.

• Microservices: For large, scalable systems.

• Event-driven / Serverless: In modern cloud-based systems.

5.3. Technical Choices

• Frontend: React, Vue, Angular

• Backend: Node.js, Python (Django/FastAPI), Java (Spring), .NET Core

• Database: PostgreSQL, MongoDB, Redis

• CI/CD: GitHub Actions, Jenkins

• Cloud: AWS, Azure, GCP
• Container: Docker, Kubernetes

6. Yazılım Geliştirme Süreci (DevOps Entegrasyonu)

- Kod versiyon kontrolü (Go)
- Sürekli Entegrasyon / Dağıtım (CI/CD)
- Test otomasyonu (unit, integration, e2e)
- Monitoring (Grafana, Prometheus, ELK stack)

6. Software Development Process (DevOps Integration)
• Code version control (Go)
• Continuous Integration / Deployment (CI/CD)
• Test automation (unit, integration, e2e)
• Monitoring (Grafana, Prometheus, ELK stack)

7. Dokümantasyon Hazırlama ve Sürekli İyileştirme

- **Teknik dokümantasyon:** API, modül, kod açıklamaları
- **Kullanıcı dokümantasyonu:** Rehberler, wiki
- **Retrospektif toplantılar:** “Neyi iyi yaptık? Neyi geliştirebiliriz?”

7. Documentation Preparation and Continuous Improvement
• Technical documentation: API, module, code descriptions
• User documentation: Guides, wiki
• Retrospective meetings: “What did we do well? What can we improve?”

8. İletişim ve Kültür

- Açık iletişim, saygılı tartışma kültürü
- Ekip içinde düzenli demo’lar
- Başarıları paylaşmak, hatalardan ders çıkarmak ve öğrenmek

8. Communication and Culture
• Open communication and a culture of respectful discussion
• Regular demos within the team
• Sharing successes, learning from mistakes, and learning

Somut, profesyonel bir örnek proje modeli hazırlayalım.
Proje Adı: SmartMeet AI
Amaç: Toplantıları otomatik olarak organize eden, ses kayıtlarını metne çevirip özetleyen, toplantıdan çıkan görevleri ve beklentileri analiz eden yapay zekâ destekli bir uygulama geliştirmek.

1. PROJENİN TANIMI ve AMACI

SmartMeet AI, toplantılarda konuşulanları yapay zekâ ile analiz ederek:

- Katılımcıların söylediklerini **metne dönüştürür (ASR - Automatic Speech Recognition)**
- Metinden **özet, görev listesi ve aksiyon maddeleri** çıkarır
- Takvim ve e-posta sistemleriyle **entegrasyon** sağlar
- Ekibe **toplantı sonrası rapor** gönderir

Hedef kullanıcılar: Şirket yöneticileri, ekip liderleri, online toplantı yapan profesyoneller.

2. EKİP YAPISI ve ROLLER

Rol	**Sorumluluklar**	**Önerilen Kişi Profili**
Product Owner (PO)	Proje vizyonu, kullanıcı ihtiyaçlarını belirleme, önceliklendirme	İş Analisti veya Yönetici
Scrum Master / PM	Sprint planlaması, görev takibi, engel kaldırma	Proje Yöneticisi
Tech Lead / Software Architect	Teknik kararlar, mimari tasarım, kod kalitesi	Kıdemli Yazılımcı
Backend Developer	API’ler, veri işleme, entegrasyon	Python (FastAPI) veya Node.js uzmanı
Frontend Developer	UI/UX, React arayüzü, dashboard tasarımı	React/Next.js uzmanı
AI/ML Engineer	Ses-metin analizi, özetleme, duygu analizi	NLP / OpenAI Whisper bilgisi
QA Engineer	Test senaryoları, otomatik testler	Test otomasyon uzmanı
UI/UX Designer	Arayüz tasarımı, kullanıcı deneyimi	Figma / Adobe XD uzmanı
DevOps Engineer	CI/CD, Docker, AWS / GCP kurulumu	Cloud uzmanı

3. PROJE METODOLOJİSİ: AGILE - SCRUM

Proje 4 haftalık **sprint** döngüleriyle yönetilecek.

Her sprint sonunda:

- Çalışan bir ürün parçası sunulur (örneğin: “otomatik konuşma tanıma” modülü)
- Demo toplantısı yapılır
- Retrospektif ile iyileştirme önerileri alınır

Araçlar:

- Jira (task yönetimi)
- Notion / Confluence (dokümantasyon)
- Slack (iletişim)
- GitHub (kod yönetimi)

4. PROJE YOL HARİTASI (ROADMAP)

Sprint	Süre	Hedef
Sprint 1 (Analiz & Mimari)	2 hafta	Gereksinimler, kullanıcı hikayeleri, mimari tasarım
Sprint 2 (Ses Tanıma)	3 hafta	Whisper tabanlı ASR modülü, ses kaydı ve metin dönüşümü
Sprint 3 (Özetleme & NLP)	3 hafta	OpenAI model entegrasyonu, metin özetleme ve görev çıkarımı
Sprint 4 (UI / Dashboard)	3 hafta	React arayüz, kullanıcı oturumu, rapor görüntüleme
Sprint 5 (Test & Dağıtım)	2 hafta	Testler, hata düzeltme, CI/CD kurulumu
Sprint 6 (Kapanış & Dokümantasyon)	1 hafta	Kullanıcı rehberi, sunum, lansman

Toplam: ~14 hafta (yaklaşık 3,5 ay)

5. TEKNİK MİMARİ TASARIMI

◆ **Genel Yapı:**

[Client - React/Next.js]

[API Gateway - FastAPI / Node.js]

[AI Services Layer]
├── Speech-to-Text (Whisper)
├── NLP Engine (OpenAI GPT API)
└── Sentiment / Task Extractor

[Database Layer]
├── PostgreSQL (veri)
├── Redis (cache)
└── S3 / GCS (ses dosyaları)

[DevOps / Cloud]
├── Docker + Kubernetes
├── AWS EC2 / GCP Compute
└── CI/CD: GitHub Actions

6. MİMARİ İLKELER

- **Katmanlı yapı:** Arayüz – API – İş Mantığı – Veri
- **Microservice yaklaşımı:** AI servisleri bağımsız modüller
- **Scalability:** Bulut tabanlı container altyapısı
- **Security:** JWT authentication, HTTPS zorunluluğu
- **Observability:** Logging + Monitoring (Grafana + Prometheus)

7. TEST VE KALİTE YÖNETİMİ

- **Unit test:** PyTest / Jest
- **Integration test:** Postman / Newman
- **E2E test:** Cypress
- **Code review:** Her PR'da 2 onay
- **Continuous Testing:** CI/CD sürecine bağlı

8. DOKÜMANTASYON

- **Teknik:** API doc (Swagger / OpenAPI)
- **Kullanıcı:** Wiki / PDF rehber
- **Yönetim:** Sprint raporları, risk tablosu

9. SÜREKLİ GELİŞİM

- Her sprint sonunda retro toplantısı yapılır.
- Ölçümler: Velocity, bug oranı, kullanıcı geri bildirimi.
- Öğrenilen dersler sonraki sprint planına entegre edilir.

10. SONUÇ

Bu yapı, küçük bir ekiple (6–8 kişi) 3–4 ay içinde çalışan bir **MVP (Minimum Viable Product)** oluşturmanı sağlar.

MVP sonrasında kullanıcı geri bildirimleriyle yeni özellikler (örneğin: gerçek zamanlı çeviri, duygu analizi, görev yönetimi entegrasyonu) eklenebilir.

8. ToplantıAI — Yapay Zekâ Destekli Toplantı Yönetim Uygulaması

Amaç: Toplantıları otomatik organize eden; toplantı ses kayıtlarını yüksek doğrulukla metne çeviren (ASR); toplantı özetleri, alınacak aksiyonlar, sorumlular ve beklentileri çıkaran; görev takibi için çıktılar üreten bir yazılımın tasarımı, MVP planı ve uygulanabilir teknik yol haritası.

1. Hızlı Özet (TL;DR)

- Kullanıcı takvimiyle entegre olarak toplantı davetleri oluşturur ve hatırlatır.
- Toplantı sırasında/sonrasında ses kaydını alır ve otomatik olarak metne çevirir.
- Toplantı metininden özet, kararlar, aksiyon maddeleri, sahipleri ve teslim tarihleri çıkarır.
- Çıktıları görev yönetimi sistemine (ör. Trello, Asana, Jira) otomatik olarak iletebilir.
- Güvenlik ve gizlilik: uçtan uca şifreleme, veri saklama politikaları, izin yönetimi.

2. Hedef Kullanıcılar

- KOBİ'ler, proje ekipleri, ürün ekipleri, danışmanlık firmaları ve uzaktan çalışan ekipler.
- İleri kullanım: kurumsal (on-prem veya VPC içinde çalıştırma) ve SaaS.

3. Ana Özellikler

1. **Takvim Entegrasyonu** (Google Calendar, Microsoft 365)
    - Otomatik davet oluşturma, uygunluk kontrolü, toplantı odası/bağlantı ekleme.
2. **Toplantı Otomasyonu**
    - Otomatik join (Zoom/Teams), veya ses bağlantısı (VoIP) üzerinden kaydetme.
    - Canlı altyazı / gerçek zamanlı transkripsiyon (opsiyonel).
3. **ASR (Speech-to-Text)**
    - Yüksek doğruluklu Türkçe ve diğer diller.
    - Speaker diarization (kim ne söylediğini ayırma).
4. **Toplantı Özetleme & Bilgi Çıkarma**
    - Kısa özet (1–3 cümle), detaylı özet (paragraf).
    - Aksiyon maddeleri: görev, sorumlu, teslim tarihi, öncelik.
    - Kararlar, açık sorular, riskler, takip maddeleri.
5. **Görev Yönetimi**
    - Dahili görev panosu + entegrasyon (Trello, Asana, Jira, Slack).
6. **Arama & Arşiv**

    - Toplantı kayıtlarında tam metin arama, konuşmacı bazlı filtreler, etikete göre filtre.
7. **Güvenlik ve Uyumluluk**
    - Kayıt veri yaşam döngüsü (retention), kullanıcı izinleri, E2E şifreleme opsiyonu.

8. **Yönetici Paneli & Analitik**
    - Toplantı sayıları, ortalama aksiyon sayısı, tamamlanma oranları.

4. Yüksek Seviyeli Mimari

Bileşenler:

- **Frontend**: React (Tercih: Tailwind, component library)
- **Backend API**: REST/GraphQL (Node.js/Typescript veya Python/FastAPI)
- **ASR Servisi**: Open-source (whisperX/whisper) veya bulut (Google Speech-to-Text, Azure Speech, Amazon Transcribe)
- **LLM/NLU**: Açık modeller veya API (OpenAI / Azure OpenAI / yerel LLM). Metin özetleme, bilgi çıkarımı için LLM promptları.
- **Realtime**: WebSocket / WebRTC (canlı altyazı, canlı join)
- **Datastore**: Postgres (meta), S3-compatible obje depolama (ses dosyaları, transcript ham verisi)
- **Queue**: RabbitMQ / Redis Streams / SQS (işlem kuyruğu: ASR, NLU işler için)
- **Integrations**: OAuth + API clients (Google Calendar, MS Graph, Zoom, Slack, Trello)
- **Auth / IAM**: OAuth2.0, JWT, rol tabanlı erişim

5. Veri Akışı (Örnek)

1. Kullanıcı takvimden veya uygulama içinden toplantı planlar.
2. Toplantı başladığında sistem otomatik bağlanır ve ses kaydını alır; kayıt S3'e gider.
3. Kayıt tamamlanınca ASR iş kuyruğuna görev gönderilir.
4. ASR çıktısı (raw transcript + speaker timestamps) elde edilir.
5. Transcript LLM/NLU pipeline'a gider: özetleme, aksiyon çıkarımı, rol/tarih tespiti.
6. Çıktılar veritabanına kaydedilir ve kullanıcıya bildirim/sunum yapılır.
7. Opsiyonel: Görevler entegre görev yöneticilerine push edilir.

6. Teknik Detaylar ve Öneriler

ASR Seçimi

- **MVP için öneri:** OpenAI Whisper (local veya API) ya da Google Speech-to-Text (yüksek doğruluk, diarization). Eğer gizlilik kritikse, on-prem Whisper türevleri tercih edin.
- **Speaker diarization:** x-vector tabanlı diarization (pyannote.audio) veya bulut diarization.

NLU / Özetleme

- **Yaklaşım:** İki aşamalı pipeline
    1. Kural-temelli çıkarımlar (regex/date parsing, isim tanıma) + Heuristics
    2. LLM ile semantic çıkarım (promptlar: özet, aksiyon maddesi çıkar, sorumlu bul)
- **Prompt örnekleri:** (Dokümanda örnek prompt kümesi verilecek)

Veri Şeması (kısa)

- meetings (id, title, start_time, end_time, organizer_id, calendar_event_id, recording_url, transcript_id)
- transcripts (id, meeting_id, raw_text, speakers_json, timestamps)
- actions (id, meeting_id, description, owner_id, due_date, status, confidence)
- users, integrations, audit_logs

API Örnekleri

- POST /api/v1/meetings — toplantı oluştur
- GET /api/v1/meetings/{id}/summary — toplantı özeti ve aksiyonlar
- POST /api/v1/meetings/{id}/recording — kayıt yükle
- GET /api/v1/actions?user_id=... — kullanıcı görevleri

---

7. MVP Tanımı (Önceliklendirilmiş)

Sprint 0 — Hazırlık ve altyapı

- Takvim OAuth entegrasyonu (Google), temel auth
- Basit veritabanı + obje depolama

Sprint 1 — Kayıt & ASR

- Manuel kayıt yükleme + ASR pipeline (Whisper via API)
- Transcript storage ve viewer

Sprint 2 — Özetleme & Aksiyon Çıkarma

- Basit LLM prompt tabanlı özetleme
- Regex/tabanlı görev çıkarımı + sahip eşleme

Sprint 3 — Otomasyon & Entegrasyonlar

- Toplantı otomatik join (Zoom/Teams) veya toplantı bağlantısı yönetimi
- Görev yöneticilerine push (Trello/Asana) + Slack bildirimleri

Sprint 4 — Canlı altyazı & Diyalog analizi

- Gerçek zamanlı transkripsiyon (opsiyonel)

- Konuşmacı duygusal tonu, konuşma dağılımı analitiği

8. Örnek Promptlar (LLM için)

- **Kısa özet:**

Verilen toplantı transkriptini 2-3 cümle ile özetle. Özet kısa ve eylem odaklı olsun.

Transkript:
"<TRANSCRIPT>"

- **Aksiyon çıkarma:**

Transkript içinden açık aksiyon maddelerini çıkar. Her aksiyon için: açıklama, muhtemel sorumlu (isim bulunuyorsa ata), önerilen son teslim tarihi (eğer tarih söz konusuysa), ve güven skorunu ver.

- **Karar ve risk çıkarımı:** benzer şekilde.

9. Gizlilik, Güvenlik ve Uyumluluk

- Varsayılan: kayıtlar şifreli olarak saklanır (server-side encryption). Kritik müşteriler için E2E (client-side) encryption seçeneği.
- Veri saklama politikası: varsayılan 90 gün, yönetici ayarı ile değiştirilebilir.
- GDPR/TKV uyumluluğu için veri silme, veri erişim kayıtları, IAM.

10. Maliyet Tahmini (kabaca)

- ASR maliyeti (API kullanılıyorsa) => ses dakikası başına değişir.
- LLM çağrıları => token bazlı maliyet.
- Depolama & deploy => S3 + DB + sunucu maliyeti. (Bu kısım müşteriye ve kullanılan servislere göre kesinleştirilir.)

11. Test & Doğrulama

- ASR doğruluğu: WER metric (kelime hata oranı), konuşmacı ayrımı doğruluğu.
- NLU doğruluğu: precision/recall aksiyon çıkarımı.
- Kullanıcı testi: pilot ekiplerle 2 haftalık deneme.

12. Geliştirme Zaman Çizelgesi (Örnek)

- MVP: 8–12 hafta (2–3 yazılımcı, 1 ML müh., 1 ürün yöneticisi)
- Kurumsal hazır ürün: +8–12 hafta entegrasyon, güvenlik sertifikaları.

13. İleri Özellik Önerileri

- Toplantı sırasında not alma asistanı (öneriler sunan, soruları hatırlatan)
- Çokdilli toplu çeviri + çoklu altyazı
- Otomatik tutanak oluşturma, imza hali
- KPI’lar için düzenli raporlar (haftalık/aylık)

14. Ek: Hızlı Yol Haritası — İlk 2 Haftada Ne Yapılır?

1. Teknik keşif: kullanılacak ASR ve LLM kararını verin.
2. Basit prototype: kullanıcı takviminden toplantı al, manuel ses kaydı yükle, Whisper ile transcript al, LLM ile kısa özet üret.
3. Pilot: 1–2 ekip ile test, geri bildirim topla.

15. Ek Materyaller

- Örnek API endpoint'leri, DB migration şeması ve örnek prompt'lar proje ilerledikçe eklenecektir.

9. UML (Unified Modeling Language) diagrams

UML (Unified Modeling Language) **diagrams** are standardized visual representations used in **software and systems design** to model the structure and behavior of systems. They help developers, analysts, and stakeholders understand, design, and document software systems clearly.

� UML Diagram Categories

UML diagrams are generally divided into **two main categories**:

1. Structural Diagrams

Describe the **static aspects** of a system — what components exist and how they are related.

Diagram Type	Description	Common Use
Class Diagram	Shows classes, attributes, methods, and relationships (inheritance, association).	Object-oriented design
Object Diagram	Instance-level snapshot of objects and their relationships.	Testing data models
Component Diagram	Describes software components and dependencies.	System architecture
Deployment Diagram	Shows physical deployment of software (servers, nodes).	System infrastructure
Package Diagram	Organizes classes into packages or modules.	High-level organization
Composite Structure Diagram	Shows internal structure of a class/component.	Advanced OO modeling

2. Behavioral Diagrams

Describe the **dynamic aspects** of a system — how objects interact and change over time.

Diagram Type	Description	Common Use
Use Case Diagram	Illustrates system functionality from the user's perspective (actors & use cases).	Requirements analysis
Sequence Diagram	Shows object interactions in time sequence (messages between objects).	Scenario modeling
Activity Diagram	Models workflow or business process logic (like a flowchart).	Process modeling
State Machine Diagram	Describes states and transitions of an object.	Modeling complex state changes
Communication Diagram	Similar to sequence diagram, but focuses on object relationships.	Interaction overview
Interaction Overview Diagram	Combines sequence and activity diagrams for complex workflows.	High-level behavior
Timing Diagram	Shows change of state or condition over time.	Real-time systems