ORM

An ORM (Object-Relational Mapper) is used to interact with a database using an object-oriented programming language. ORMs allow developers to work with databases using familiar, object-oriented concepts, rather than writing raw SQL statements.

- This can make the development process more efficient and less error-prone, as well as allow for easier maintenance of the codebase.
- Additionally, ORMs often provide features such as caching, lazy loading, and connection pooling, which can improve application performance.
- They also provide an abstraction layer between the application and the database, so that the application code can be insulated from changes to the underlying database schema.
- This can make it easier to switch to a different database in the future or to improve scalability by distributing the data across multiple servers.
- Overall, ORMs can help improve developer productivity, code maintainability, and application performance.

Entity Framework Core (EF Core) is an ORM (Object-Relational Mapping) framework for the .NET platform:

Cross-platform: EF Core can be used on a variety of platforms including Windows, Linux, and Mac.

Inheritance: Support for TPC, TPH, and TPT

Support for multiple databases: EF Core supports a wide range of relational databases including SQL Server, MySQL, SQLite, and PostgreSQL.

Support for Client-evaluation and Lazy Loading.

Code first: EF Core allows developers to create a database from code, which enables a more agile and test-driven development workflow.

Lightweight: EF Core has a smaller footprint and fewer dependencies than the full version of Entity Framework.

LINQ support: EF Core supports LINQ, a powerful and expressive query language, which allows developers to write efficient and readable queries using C# or Visual Basic.

Support for explicit loading, change tracking, and caching.

Migrations: EF Core has built-in support for creating and managing database migrations, which allows for easy management of database changes over time.

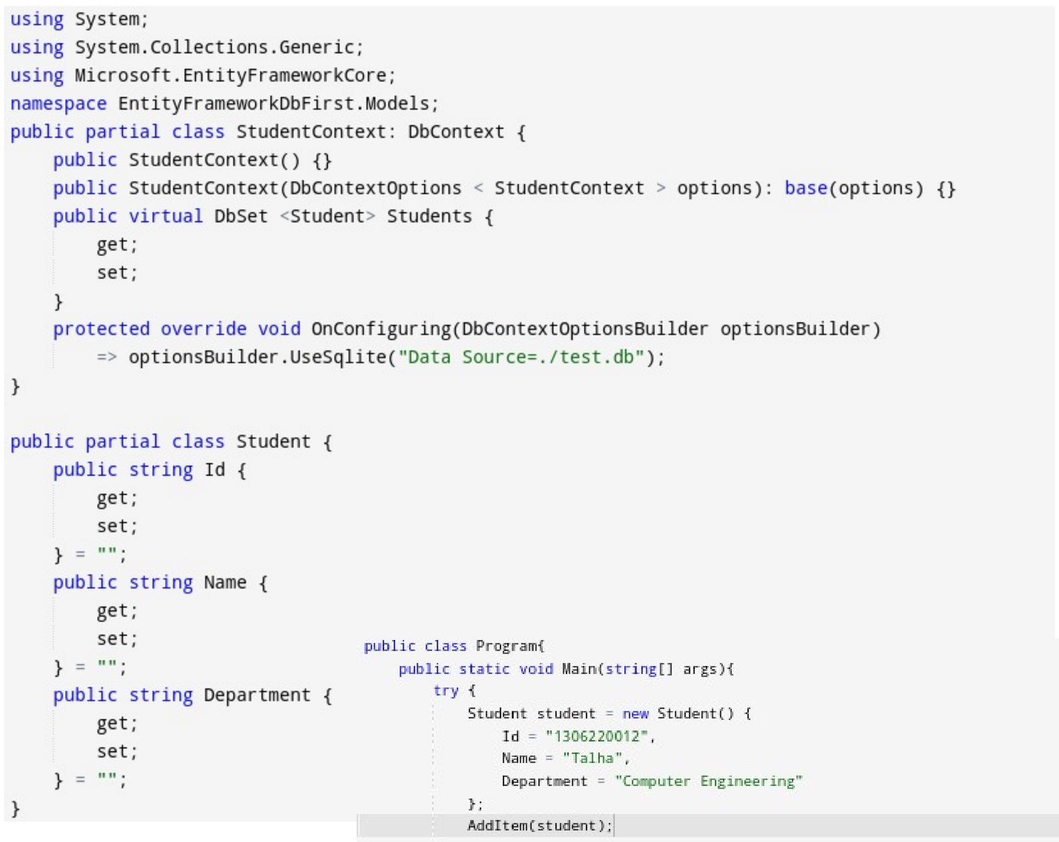
Performance improvements: EF Core has been optimized for performance and can handle large datasets efficiently.

Modeling: Support for Complex Type and Owned Type.

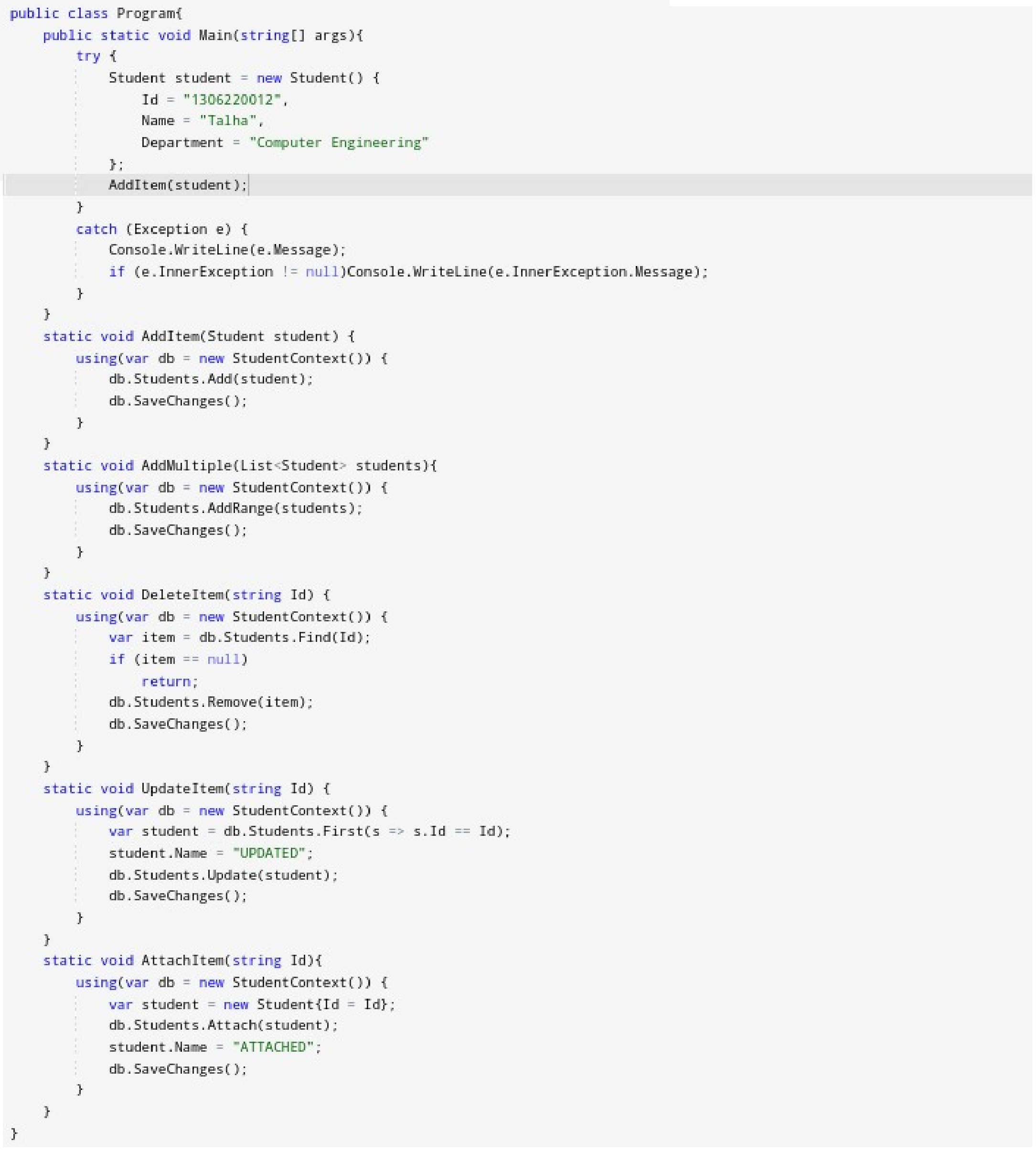
Relationships: Support for one-to-one, one-to-many, and many-to-many

Relationships: Support for one-to-one, one-to-many, and many-to-many

Example Code:



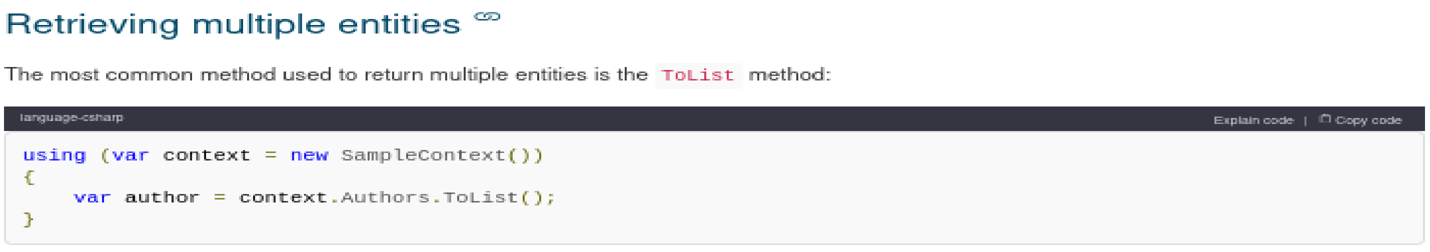
```
using System;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
namespace EntityFrameworkDbFirst.Models;
public partial class StudentContext: DbContext {
    public StudentContext() {}
    public StudentContext(DbContextOptions < StudentContext > options): base(options) {}
    public virtual DbSet <Student> Students {
        get;
        set;
    }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        => optionsBuilder.UseSqlite("Data Source=./test.db");
}

public partial class Student {
    public string Id {
        get;
        set;
    } = "";
    public string Name {
        get;
        set;
    } = "";
    public string Department {
        get;
        set;
    } = "";
}
```



```
public class Program{
    public static void Main(string[] args){
        try {
            Student student = new Student() {
                Id = "1306220012",
                Name = "Talha",
                Department = "Computer Engineering"
            };
            AddItem(student);
        }
        catch (Exception e) {
            Console.WriteLine(e.Message);
            if (e.InnerException != null)Console.WriteLine(e.InnerException.Message);
        }
    }
    static void AddItem(Student student) {
        using(var db = new StudentContext()) {
            db.Students.Add(student);
            db.SaveChanges();
        }
    }
    static void AddMultiple(List<Student> students){
        using(var db = new StudentContext()) {
            db.Students.AddRange(students);
            db.SaveChanges();
        }
    }
    static void DeleteItem(string Id) {
        using(var db = new StudentContext()) {
            var item = db.Students.Find(Id);
            if (item == null)
                return;
            db.Students.Remove(item);
            db.SaveChanges();
        }
    }
    static void UpdateItem(string Id) {
        using(var db = new StudentContext()) {
            var student = db.Students.First(s => s.Id == Id);
            student.Name = "UPDATED";
            db.Students.Update(student);
            db.SaveChanges();
        }
    }
    static void AttachItem(string Id){
        using(var db = new StudentContext()) {
            var student = new Student{Id = Id};
            db.Students.Attach(student);
            student.Name = "ATTACHED";
            db.SaveChanges();
        }
    }
}
```

Other Samples:

- Retrieve Multiple Entities

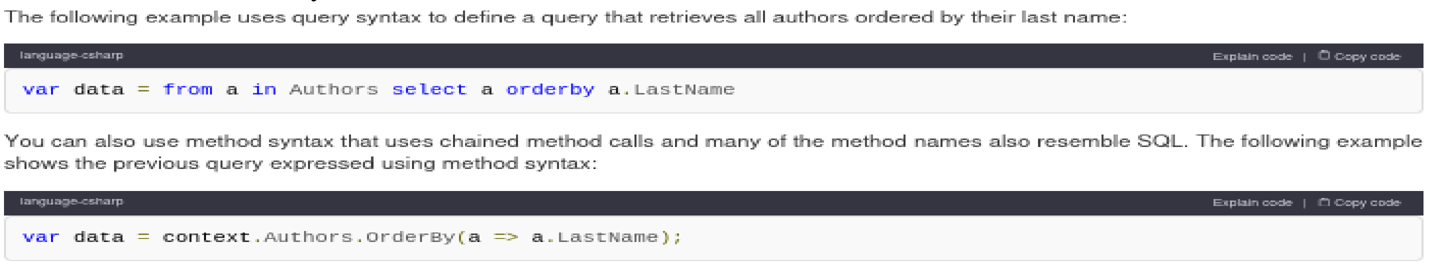


Retrieving multiple entities

The most common method used to return multiple entities is the `ToList` method:

```csharp
using (var context = new SampleContext())
{
    var author = context.Authors.ToList();
}
```

- OrderBy



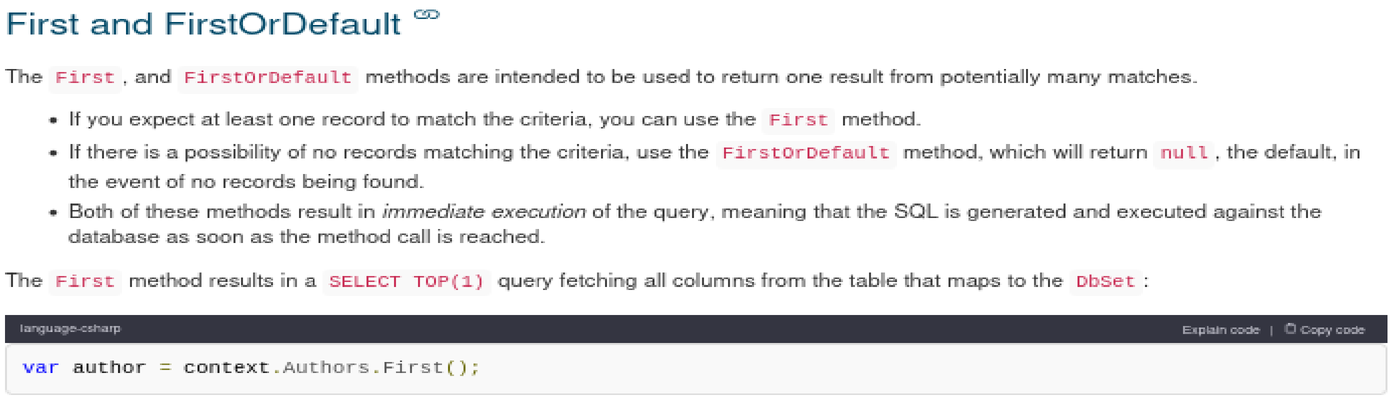
The following example uses query syntax to define a query that retrieves all authors ordered by their last name:

```csharp
var data = from a in Authors select a orderby a.LastName
```

You can also use method syntax that uses chained method calls and many of the method names also resemble SQL. The following example shows the previous query expressed using method syntax:

```csharp
var data = context.Authors.OrderBy(a => a.LastName);
```

- Retrieve the first element



First and FirstOrDefault

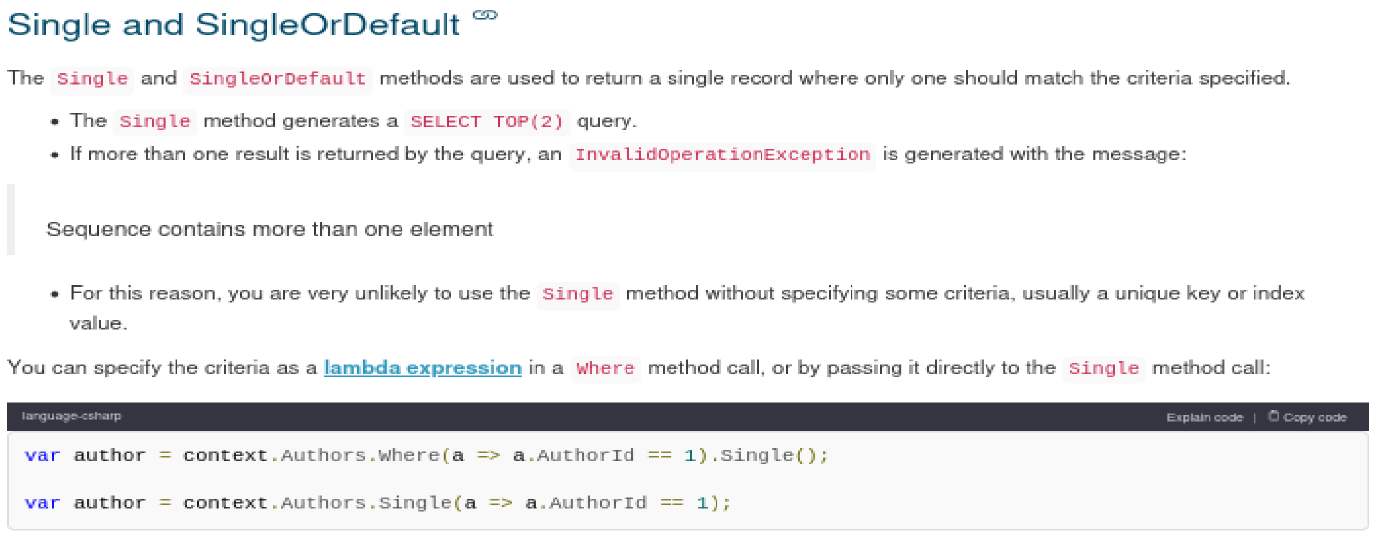
The `First`, and `FirstOrDefault` methods are intended to be used to return one result from potentially many matches.

- If you expect at least one record to match the criteria, you can use the `First` method.
- If there is a possibility of no records matching the criteria, use the `FirstOrDefault` method, which will return `null`, the default, in the event of no records being found.
- Both of these methods result in *immediate execution* of the query, meaning that the SQL is generated and executed against the database as soon as the method call is reached.

The `First` method results in a `SELECT TOP(1)` query fetching all columns from the table that maps to the `DbSet`:

```csharp
var author = context.Authors.First();
```

- Retrieve only one element



Single and SingleOrDefault

The `Single` and `SingleOrDefault` methods are used to return a single record where only one should match the criteria specified.

- The `Single` method generates a `SELECT TOP(2)` query.
- If more than one result is returned by the query, an `InvalidOperationException` is generated with the message:

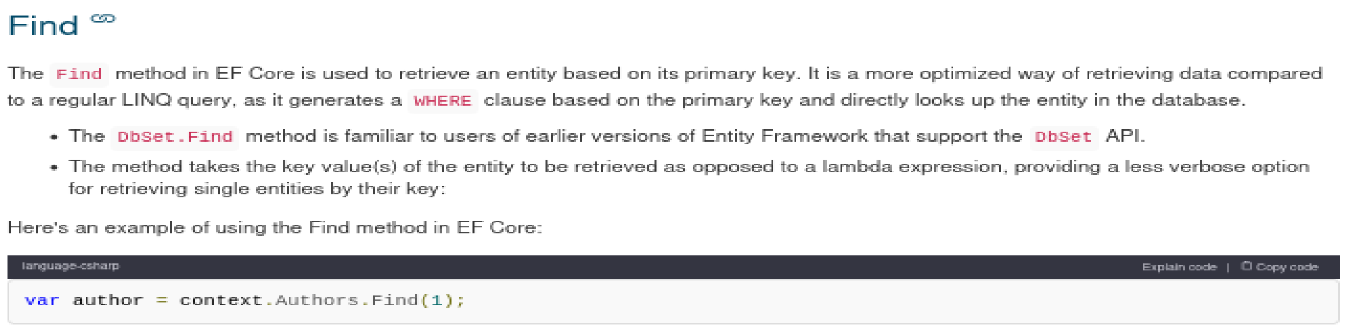
> Sequence contains more than one element

- For this reason, you are very unlikely to use the `Single` method without specifying some criteria, usually a unique key or index value.

You can specify the criteria as a lambda expression in a `Where` method call, or by passing it directly to the `Single` method call:

```csharp
var author = context.Authors.Where(a => a.AuthorId == 1).Single();

var author = context.Authors.Single(a => a.AuthorId == 1);
```

- Find and element with primary key



Find

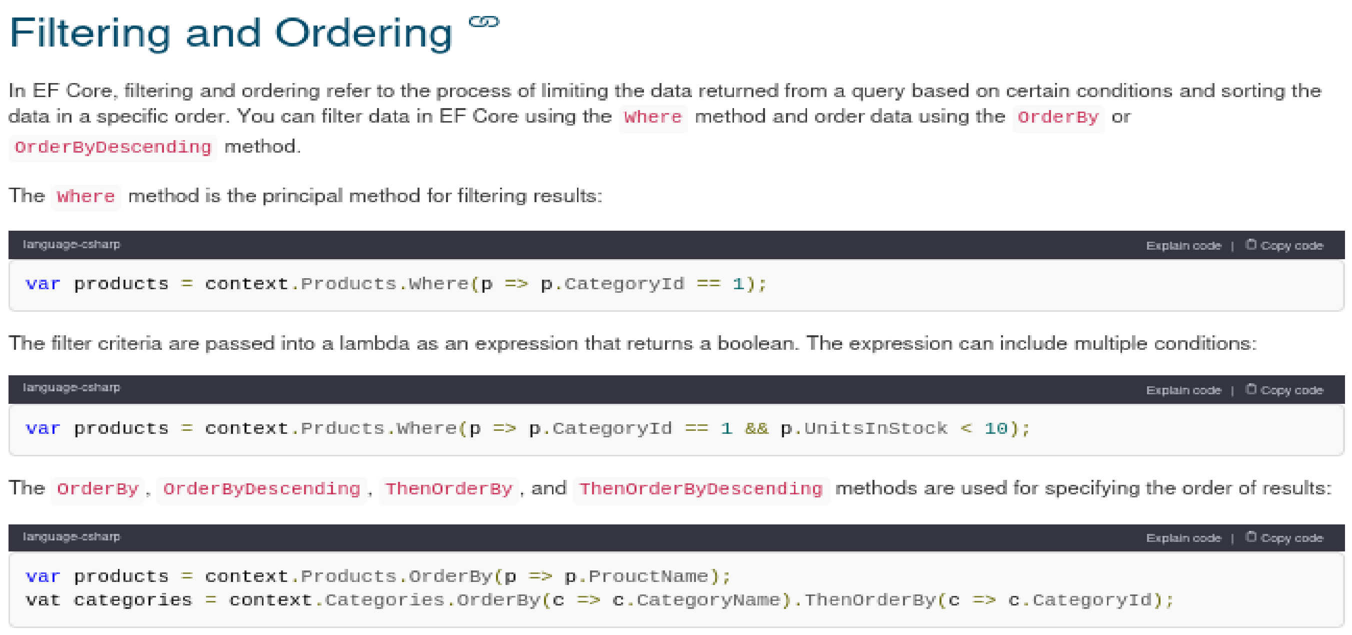
The `Find` method in EF Core is used to retrieve an entity based on its primary key. It is a more optimized way of retrieving data compared to a regular LINQ query, as it generates a `WHERE` clause based on the primary key and directly looks up the entity in the database.

- The `DbSet.Find` method is familiar to users of earlier versions of Entity Framework that support the `DbSet` API.
- The method takes the key value(s) of the entity to be retrieved as opposed to a lambda expression, providing a less verbose option for retrieving single entities by their key:

Here's an example of using the Find method in EF Core:

```csharp
var author = context.Authors.Find(1);
```

- Filter with WHERE clause and OrderBy



Filtering and Ordering

In EF Core, filtering and ordering refer to the process of limiting the data returned from a query based on certain conditions and sorting the data in a specific order. You can filter data in EF Core using the `Where` method and order data using the `OrderBy` or `OrderByDescending` method.

The `Where` method is the principal method for filtering results:

```csharp
var products = context.Products.Where(p => p.CategoryId == 1);
```

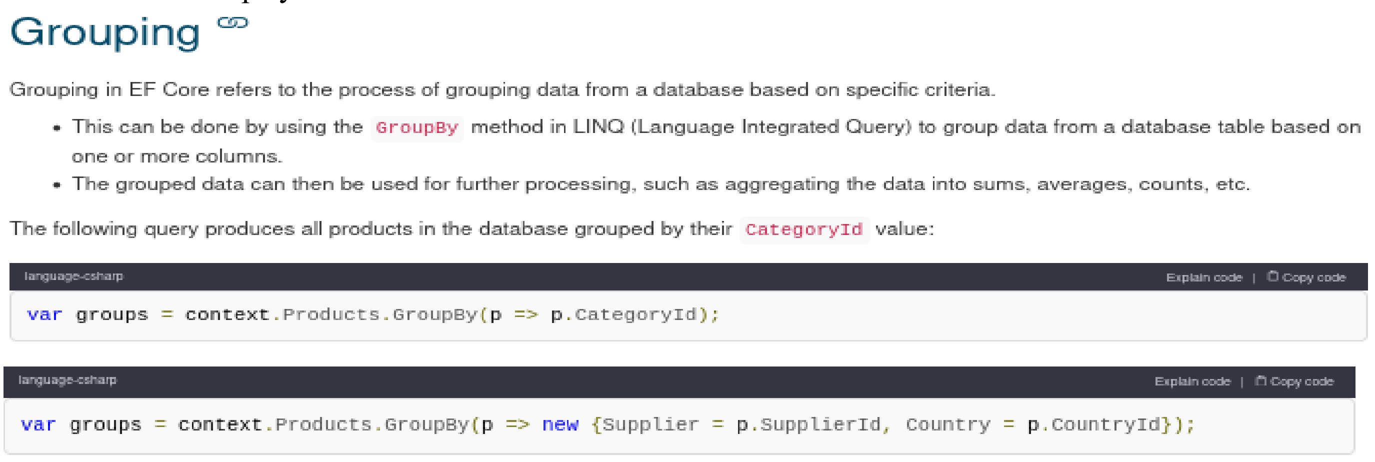
The filter criteria are passed into a lambda as an expression that returns a boolean. The expression can include multiple conditions:

```csharp
var products = context.Prducts.Where(p => p.CategoryId == 1 && p.UnitsInStock < 10);
```

The `OrderBy`, `OrderByDescending`, `ThenOrderBy`, and `ThenOrderByDescending` methods are used for specifying the order of results:

```csharp
var products = context.Products.OrderBy(p => p.ProuctName);
vat categories = context.Categories.OrderBy(c => c.CategoryName).ThenOrderBy(c => c.CategoryId);
```

- GroupBy



Grouping

Grouping in EF Core refers to the process of grouping data from a database based on specific criteria.

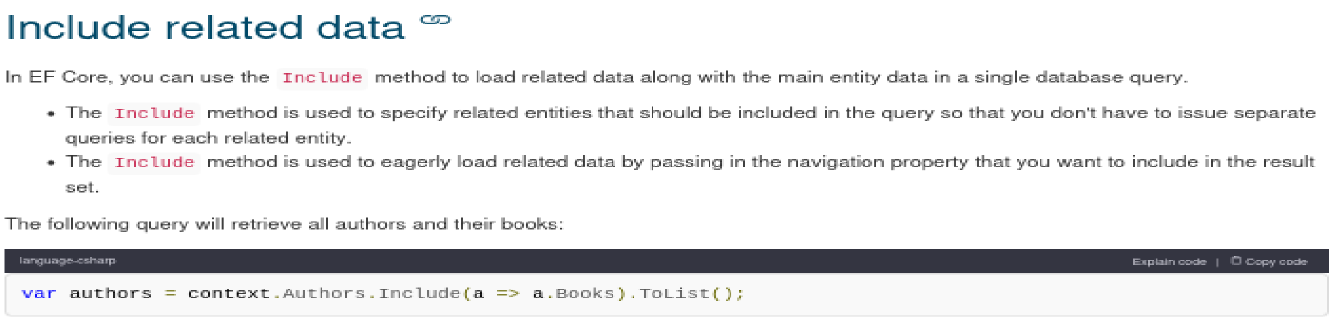
- This can be done by using the `GroupBy` method in LINQ (Language Integrated Query) to group data from a database table based on one or more columns.
- The grouped data can then be used for further processing, such as aggregating the data into sums, averages, counts, etc.

The following query produces all products in the database grouped by their `CategoryId` value:

```csharp
var groups = context.Products.GroupBy(p => p.CategoryId);
```

```csharp
var groups = context.Products.GroupBy(p => new {Supplier = p.SupplierId, Country = p.CountryId});
```

- Include related data inside query



Include related data

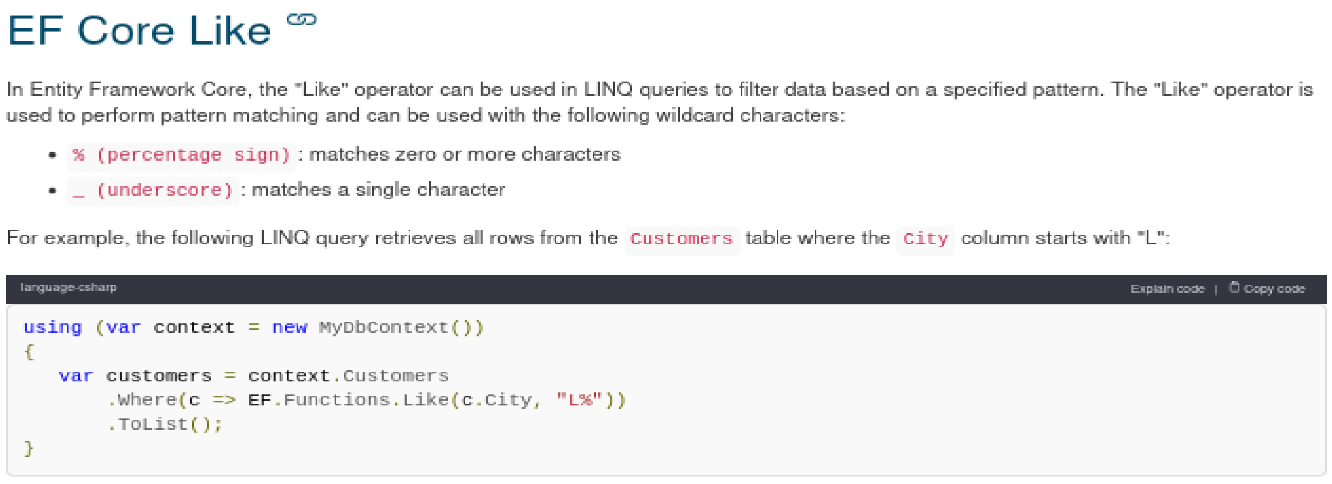
In EF Core, you can use the `Include` method to load related data along with the main entity data in a single database query.

- The `Include` method is used to specify related entities that should be included in the query so that you don't have to issue separate queries for each related entity.
- The `Include` method is used to eagerly load related data by passing in the navigation property that you want to include in the result set.

The following query will retrieve all authors and their books:

```csharp
var authors = context.Authors.Include(a => a.Books).ToList();
```

- LIKE clause



EF Core Like

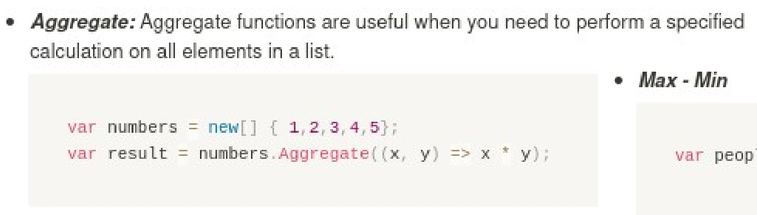
In Entity Framework Core, the "Like" operator can be used in LINQ queries to filter data based on a specified pattern. The "Like" operator is used to perform pattern matching and can be used with the following wildcard characters:

- `% (percentage sign)` : matches zero or more characters
- `_ (underscore)` : matches a single character

For example, the following LINQ query retrieves all rows from the `Customers` table where the `City` column starts with "L":

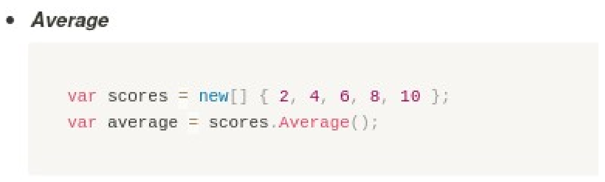
```csharp
using (var context = new MyDbContext())
{
    var customers = context.Customers
        .Where(c => EF.Functions.Like(c.City, "L%"))
        .ToList();
}
```

LINQ



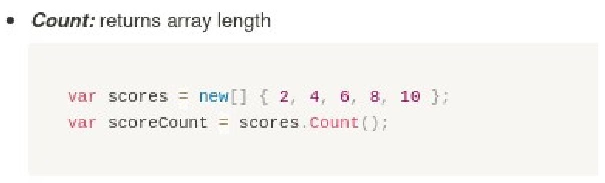
- ***Aggregate:*** Aggregate functions are useful when you need to perform a specified calculation on all elements in a list.

```csharp
var numbers = new[] { 1,2,3,4,5};
var result = numbers.Aggregate((x, y) => x * y);
```



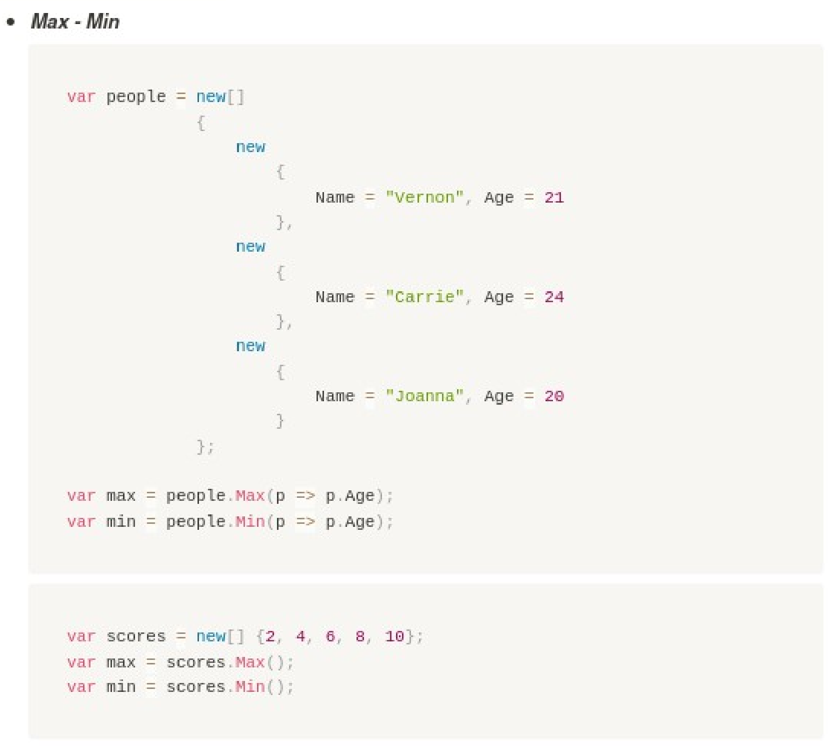
- ***Average***

```csharp
var scores = new[] { 2, 4, 6, 8, 10 };
var average = scores.Average();
```



- ***Count:*** returns array length

```csharp
var scores = new[] { 2, 4, 6, 8, 10 };
var scoreCount = scores.Count();
```



- ***Max - Min***

```csharp
var people = new[]
             {
                 new
                     {
                         Name = "Vernon", Age = 21
                     },
                 new
                     {
                         Name = "Carrie", Age = 24
                     },
                 new
                     {
                         Name = "Joanna", Age = 20
                     }
             };

var max = people.Max(p => p.Age);
var min = people.Min(p => p.Age);
```

```csharp
var scores = new[] {2, 4, 6, 8, 10};
var max = scores.Max();
var min = scores.Min();
```

- ***All:*** method is useful when you need to verify that all elements in a list satisfy a particular boolean condition.

```
var scores = new[] { 87, 56, 33, 20, 11 };
var all = scores.All(i => i > 0);
```

- ***Any:*** operator verifies that at least any one item in the list or array satisfies a particular boolean condition.

```
var scores = new[] { 87, 56, 33, 20, 11, -10, -50 };
var all = scores.Any(i => i > 10);
```

- ***Concat***

```
var set1 = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var set2 = new[] { 2, 4, 6, 8, 10 };

var result = set1.Concat(set2);
```

- ***Contains***

```
var colors = new[] { "red", "blue", "green", "yellow", "purple", "orange" };
bool result = colors.Contains("blue");
```

- ***ToDictionary***

```
var countries = new[]
    {
        new {CountryName = "United Kingdom", Code = "GB", Currency = "GBP"},
        new {CountryName = "United States", Code = "US", Currency = "USD"},
        new {CountryName = "Canada", Code = "CA", Currency = "CAD"},
        new {CountryName = "France", Code = "FR", Currency = "EUR"},
        new {CountryName = "Spain", Code = "ES", Currency = "EUR"}
    };


var results = countries.ToDictionary(arg => arg.Code, arg => arg.CountryName);
```

- ***ElementAt - ElementAtOrDefault***

```
var numbers = new[] { 2, 4, 6, 8, 10 };
int number = numbers.ElementAt(3);
int number2 = numbers.ElementAtOrDefault(13);
```

- ***First - Last***

```
var people = new[]
    {
        new
            {
                Id = 1, Name = "Vernon", Gender = "Male", CountryCode = "GB"
            },
        new
            {
                Id = 2, Name = "Carrie", Gender = "Female", CountryCode = "C
            },
        new
            {
                Id = 3, Name = "Joanna", Gender = "Female", CountryCode = "U
            },
    };

var carrie = people.First(p => p.Id == 3);
var carrie = people.Last(p => p.Id < 3);
```

```
var numbers = new[] { 2, 4, 6, 8, 10 };
int first = numbers.First();
int last = numbers.Last();
```

- ***FirstOrDefault - LastOrDefault***

```
var numbers = new[] { 2, 4, 6, 8, 10 };
int firstvalue = numbers.FirstOrDefault(n => n < 0)
int lastvalue = numbers.LastOrDefault(n => n < 0)
```

- ***Single - SingleOrDefault:*** used when you expect response to be exactly 1 record otherwise it will give error. If ***SingleOrDefault*** used and no record found it will return null or zero.

```
List<int> numbers = new List<int>() { 10, 20, 30 }; ;
int numberMS = numbers.Single(num => num == 20);
int numberMS = numbers.SingleOrDefault(num => num == 40);
```

- ***Where***

```
int[] numbers = { 40, 10, 24, 60, 100, 55, 80, 75, 35 };

var numbersAboveSeventy = numbers.Where(num => num > 70);
var numbersAboveSeventy2 = from num in numbers where num > 70 select num;
```

```
int[] set1 = { 40, 10, 24, 60, 100, 55, 80, 75, 35 };
int[] set2 = { 120, 110, 150, 160, 100, 155, 180, 175, 135 };

var results = set1.Where(i => set2.Contains(i));
```

```
var customers = new[] {
    new{
            Name = "Vernon",
            DateOfBirth = "1994-Jun-25",
            Active = true,
            Card = "MasterCard",
            CardNumer = "*** 1142",
            ExpiryYear = 2022,
            ExpiryMonth = 7
        },
    new {
            Name = "Carrie",
            DateOfBirth = "1986-Feb-01",
            Active = false,
            Card = "Visa",
            CardNumer = "*** 2156",
            ExpiryYear = 2015,
            ExpiryMonth = 7
        },
    new {
            Name = "Joanna",
            DateOfBirth = "1972/10/13",
            Active = true,
            Card = "Visa",
            CardNumer = "*** 7683",
            ExpiryYear = 2014,
            ExpiryMonth = 3
        },
};

var results = customers.Where(customer => customer.Card == "Visa"
                    && customer.ExpiryYear <= 2015);
```

- ***GroupBy***

```
var people = new[]
    {
        new
            {
                Name = "Vernon",
                Gender = "Male",
            },
        new
            {
                Name = "Carrie",
                Gender = "Female"
            },
        new
            {
                Name = "Thomas",
                Gender = "Male"
            }
    };

var peopleByGender = people.GroupBy(p => p.Gender)
                            .Select(x => new { Gender = x.Key, People = x })

var peopleByGender2 = people.GroupBy(arg => arg.Gender,
                            (key, list) => new {Gender = key, People = lis

var peopleByGender3 = from person in people
                     group person by person.Gender
                     into g
                     select new {Gender = g.Key, People = g};
```

```
//Multi parameter Grouping
var people = new[]
    {
        new
            {
                Name = "Vernon",
                Gender = "Male",
                City = "London",

            },
        new
            {
                Name = "Carrie",
                Gender = "Female",
                City = "London"
            },
        new
            {
                Name = "Joanna",
                Gender = "Female",
                City = "London"
            },
        new
            {
                Name = "Thomas",
                Gender = "Male",
                City = "Paris"
            }
    };

var peopleByGender = people.GroupBy(arg => new { arg.Gender, arg.City })
                            .Select(grouping => new
                            {
                                Gender = grouping.Key.Gender,
                                City = grouping.Key.City,
                                People = grouping
                            });

 var peopleByGender2 = from p in people
                     group p by new { p.Gender, p.City }
                     into g
                     select new
                     {
                        City = g.Key.City,
                        Gender = g.Key.Gender,
                        People = g
                     };
```

- *GroupJoin*

```
var people = new[]
    {
        new { Name = "Vernon", Gender = "Male", CountryCode = "GB" },
        new { Name = "Carrie", Gender = "Female", CountryCode = "CA" },
        new { Name = "Joanna", Gender = "Female", CountryCode = "US" },
        new { Name = "Thomas", Gender = "Male", CountryCode = "ES" },
        new { Name = "James", Gender = "Male", CountryCode = "GB" },
        new { Name = "Elly", Gender = "Female", CountryCode = "US" }
    };

var countries = new[]
    {
        new {CountryName = "United Kingdom", Code = "GB"},
        new {CountryName = "United States", Code = "US"},
        new {CountryName = "Canada", Code = "CA"},
        new {CountryName = "France", Code = "FR"},
        new {CountryName = "Spain", Code = "ES"}
    };

var results = countries.GroupJoin(people,
                        arg => arg.Code, // the key to select from countries
                        arg => arg.CountryCode, // the key to select from peop
                        (country, p) => new { country, PeopleInCountry = p });

var results2 = from c in countries
              join p in people
              on c.Code equals p.CountryCode
              into g
              select new { Country = c, PeopleInCountry = g};
```

- *OrderBy*

```
var cities = new[] {"Barcelona", "London", "Paris", "New York", "Moscow", "A

var orderedCities = cities.OrderBy(s => s);

var orderedCities2 = from city in cities
                     orderby city
                     select city;
```

- *OrderByDescending*

```
var cities = new[] { "Barcelona", "London", "Paris", "New York", "Moscow", "

var orderedCities = cities.OrderByDescending(s => s);
var orderedCities2 = from city in cities
                     orderby city descending
                     select city;
```

- *Reverse*

```
var cities = new[] { "Barcelona", "London", "Paris", "New York", "Moscow", "

var reverse = cities.Reverse();
```

- *InnerJoin*

```
var people = new[]
    {
        new { Name = "Vernon", Gender = "Male", CountryCode = "GB" },
        new { Name = "Carrie", Gender = "Female", CountryCode = "CA" },
        new { Name = "Joanna", Gender = "Female", CountryCode = "US" },
        new {Name = "Thomas", Gender = "Male", CountryCode = "ES" }
    };


var countries = new[]
    {
        new {CountryName = "United Kingdom", Code = "GB"},
        new {CountryName = "United States", Code = "US"},
        new {CountryName = "Canada", Code = "CA"},
        new {CountryName = "France", Code = "FR"},
        new {CountryName = "Spain", Code = "ES"}
    };


var results = people.Join(countries,
               arg => arg.CountryCode, // the key to join from the people list
               arg => arg.Code,        // the key to join from the country lis
               (person, country) => new { person.Name, country.CountryName });

 var results2 = from p in people
              join c in countries
              on p.CountryCode equals c.Code
              select new { Name = p.Name, Country = c.CountryName };
```

- *ThenBy:* allows to order multiple times

```
var people = new[]
                 {
                     new {Name = "Jennifer", Age = 21},
                     new {Name = "Alton", Age = 22},
                     new {Name = "Adam", Age = 27},
                     new {Name = "Milford", Age = 23},
                     new {Name = "James", Age = 23}
                 };


var orderedPeople = people.OrderBy(x => x.Name)
                          .ThenBy(x => x.Age);

var orderedPeople2 = from person in people
                     orderby person.Name , person.Age
                     select person;
```

- *ThenByDescending*

```
var people = new[]
                 {
                     new {Name = "Jennifer", Age = 21},
                     new {Name = "Alton", Age = 22},
                     new {Name = "Adam", Age = 27},
                     new {Name = "Milford", Age = 23},
                     new {Name = "James", Age = 23}
                 };


var orderedPeople = people.OrderByDescending(x => x.Name)
                          .ThenByDescending(x => x.Age);

var orderedPeople2 = from person in people
                     orderby person.Name descending, person.Age descending
                     select person;
```

- ***Skip***

```
var scores = new[] { 87, 56, 33, 20, 11 };

var newScores = scores.Skip(3); //skip first 3 elements
// OUTPUT : {20, 11}
```

- ***SkipWhile***

```
var scores = new[] { 87, 56, 33, 20, 11 };

var result = scores.SkipWhile(i => i > 50);
//output 33, 20, 11
```

- ***Take***

```
var scores = new[] { 87, 56, 33, 20, 11 };

var newScores = scores.Take(3); // take first 3 elements
// OUTPUT : {87, 56, 33}
```

- ***TakeWhile***

```
var scores = new[] { 87, 56, 33, 20, 11, 100 };

var newScores = scores.TakeWhile(i => i > 50);
//output: 87,56
```

- ***Select***

```
var people = new[]
{
   new {Id = 6000, Name = "Jennifer", Age = 21},
   new {Id = 6001, Name = "Alton", Age = 22},
   new {Id = 6003, Name = "Milford", Age = 23}
};

var ids = people.Select(person => person.Id);
var ids2 = from person in people
          select person.Id
```

```
var people = new[]
{
    new {Id = 6000, Name = "Jennifer", Age = 21, City="London", Country = "Uni
    new {Id = 6001, Name = "Alton", Age = 22, City="Paris", Country = "France"
    new {Id = 6003, Name = "Milford", Age = 23, City="New York", Country = "Un
};

var condensed = people.Select(person => new { Id = person.Id, Name = person.Na
var condensed2 = from p in people
                select new {Id = p.Id, Name = p.Name};
```

```
var people = new[]
 {
     new
         {
             Name = "Vernon",
             Age = 21,
             Email = "VernonKBilodeau@armyspy.com",
             Phone = "806-291-8721"
         },
     new
         {
             Name = "Carrie",
             Age = 22,
             Email = "CarrieJGlenn@dayrep.com",
             Phone = "617-389-2329"
         },
     new
         {
             Name = "Thomas",
             Age = 23,
             Email = "ThomasBWilliamson@jourrapide.com",
             Phone = "906-875-5259"
         }
 };
int index = 0;
var indexedPeople = people.Select((person, index) => new { Order = index++,
                                                  Name = person.Name,
                                                  Email = person.Email });
```

- ***SelectMany:*** is used to flatten a nested list or array

```
var people = new[]
{
    new
        {
            Name = "Vernon",
            Age = 21,
            Email = "VernonKBilodeau@armyspy.com",
            Phone = "806-291-8721",
            Orders = new[]
                {
                    new {OrderId = "AB231", Sku = "FB1232", Price = 9.95},
                    new {OrderId = "AA231", Sku = "FB5678", Price = 9.95},
                    new {OrderId = "BB231", Sku = "FB6366", Price = 9.95},
                }
        },
    new
        {
            Name = "Carrie",
            Age = 22,
            Email = "CarrieJGlenn@dayrep.com",
            Phone = "617-389-2329",
            Orders = new[]
                {
                    new {OrderId = "XF231", Sku = "QA123211", Price = 24.95}
                }
        },
};


// select all orders from everyone
var allOrders = people.SelectMany(person => person.Orders);
var allOrders2 = from person in people
                from orders in person.Orders
                select orders;
```

- ***Distinct***

```
var scores = new[] {87, 56, 56, 33, 33, 20, 11, 11};

var distinctScores = scores.Distinct();
```

```
var people = new[]
             {
                 new {Name = "Jennifer", City = "London"},
                 new {Name = "Alton", City = "London"},
                 new {Name = "Adam", City = "Paris"},
                 new {Name = "Milford", City = "New York"},
                 new {Name = "James", City = "Tokyo"},
                 new {Name = "Rachael", City = "Paris"},
             };

var distinctCities = people.Select(p => p.City).Distinct();
```

- ***Except***

```
var set1 = new[] { 10, 20, 30, 40, 50, 60 };
var set2 = new[] { 5, 10, 15, 20, 25, 30, 35 };

var result = set1.Except(set2);

//output: 40, 50, 60
```

- ***Intersect***

```
var set1 = new[] { 10, 20, 30, 40, 50, 60 };
var set2 = new[] { 5, 10, 15, 20, 25, 30, 35 };

var result = set1.Intersect(set2);

//output: 10, 20, 30
```

- ***Union***

```
var days1 = new[] {"Mon", "Tues", "Weds", "Thurs"};
var days2 = new[] {"Thurs", "Thurs", "Fri", "Sat", "Sun"};

var result = days1.Union(days2);

// output: Mon, Tues, Weds, Thurs, Fri, Sat, Sun
```

SDLC

The ***software development lifecycle (SDLC)*** is the cost-effective and time-efficient process that development teams use to design and build high-quality software. The goal of SDLC is to minimize project risks through forward planning so that software meets customer expectations during production and beyond. This methodology outlines a series of steps that divide the software development process into tasks you can assign, complete, and measure.

Design: In the design phase, software engineers analyze requirements and identify the best solutions to create the software. For example, they may consider integrating pre-existing modules, make technology choices, and identify development tools. They will look at how to best integrate the new software into any existing IT infrastructure the organization may have.

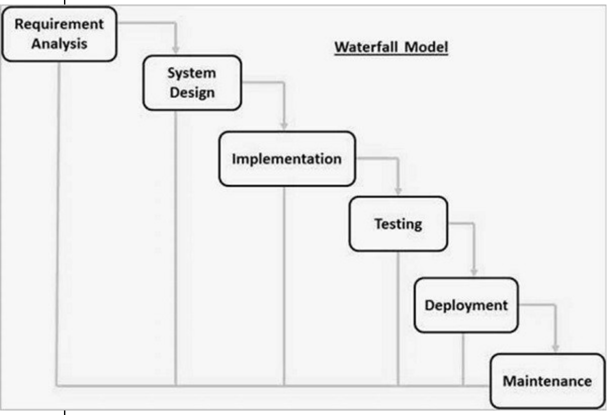
Implement: In the implementation phase, the development team codes the product. They analyze the requirements to identify smaller coding tasks they can do daily to achieve the final result.

Test: The development team combines automation and manual testing to check the software for bugs. Quality analysis includes testing the software for errors and checking if it meets customer requirements. Because many teams immediately test the code they write, the testing phase often runs parallel to the development phase.

Deploy: When teams develop software, they code and test on a different copy of the software than the one that the users have access to. The software that customers use is called *production*, while other copies are said to be in the *build environment*, or testing environment. Having separate build and production environments ensures that customers can continue to use the software even while it is being changed or upgraded. The deployment phase includes several tasks to move the latest build copy to the production environment, such as packaging, environment configuration, and installation.

Maintain: In the maintenance phase, among other tasks, the team fixes bugs, resolves customer issues, and manages software changes. In addition, the team monitors overall system performance, security, and user experience to identify new ways to improve the existing software.

Waterfall Model: The waterfall model provides discipline to project management and gives a tangible output at the end of each phase. However, there is little room for change once a phase is considered complete, as changes can affect the software's delivery time, cost, and quality. Therefore, the model is most suitable for small software development projects, where tasks are easy to arrange and manage and requirements can be pre-defined accurately.



The sequential phases in Waterfall model are −

- **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
- **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
- **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

Some of the major advantages of the Waterfall Model are as follows −

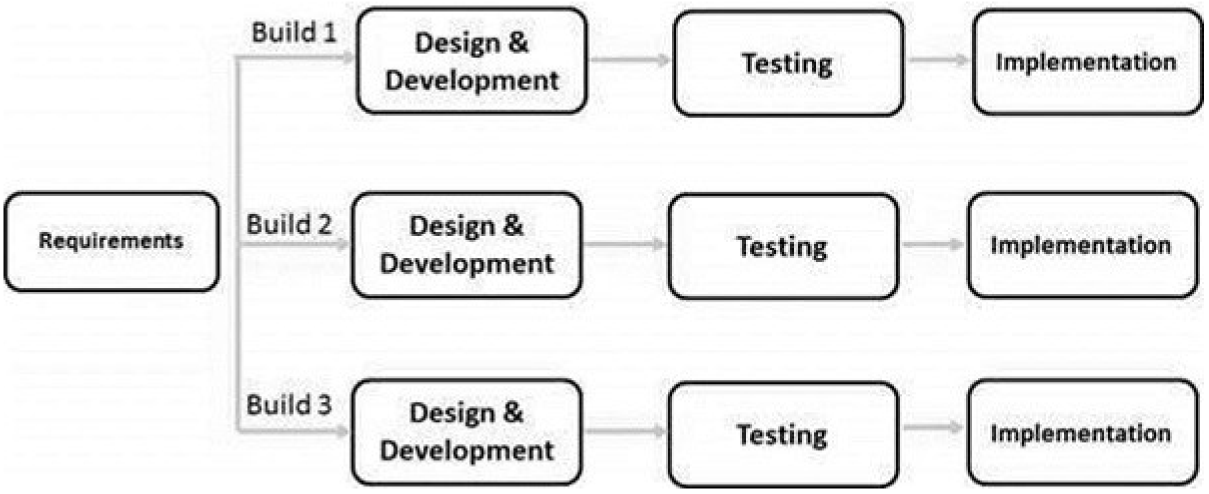
- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

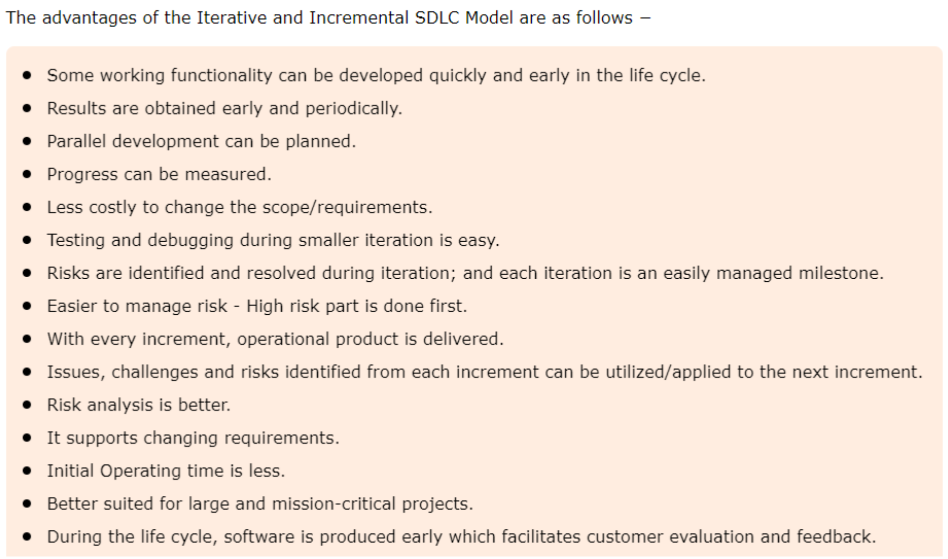
The major disadvantages of the Waterfall Model are as follows −

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- It is difficult to measure progress within stages.
- Cannot accommodate changing requirements.
- Adjusting scope during the life cycle can end a project.
- Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

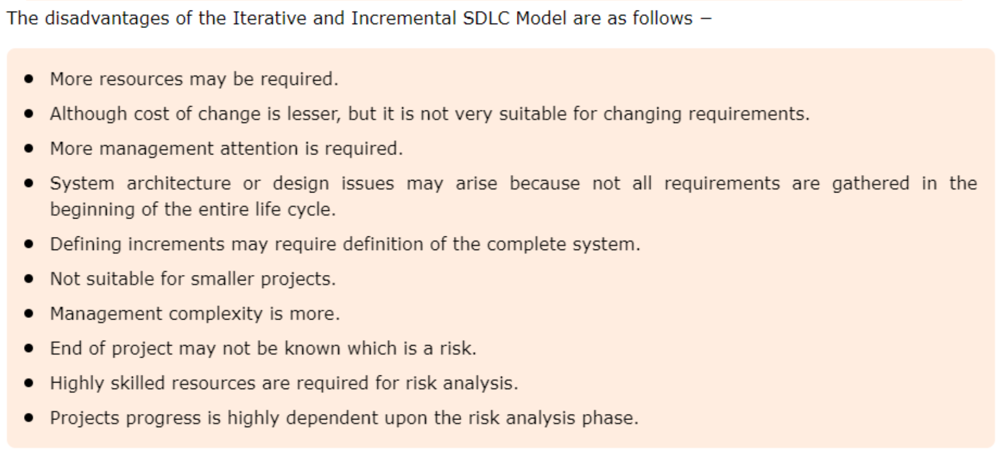
Iterative/Incremental Model: The iterative process suggests that teams begin software development with a small subset of requirements. Then, they iteratively enhance versions over time until the complete software is ready for production. The team produces a new software version at the end of each iteration.

It's easy to identify and manage risks, as requirements can change between iterations. However, repeated cycles could lead to scope change and underestimation of resources.




The advantages of the Iterative and Incremental SDLC Model are as follows –

- Some working functionality can be developed quickly and early in the life cycle.
- Results are obtained early and periodically.
- Parallel development can be planned.
- Progress can be measured.
- Less costly to change the scope/requirements.
- Testing and debugging during smaller iteration is easy.
- Risks are identified and resolved during iteration; and each iteration is an easily managed milestone.
- Easier to manage risk - High risk part is done first.
- With every increment, operational product is delivered.
- Issues, challenges and risks identified from each increment can be utilized/applied to the next increment.
- Risk analysis is better.
- It supports changing requirements.
- Initial Operating time is less.
- Better suited for large and mission-critical projects.
- During the life cycle, software is produced early which facilitates customer evaluation and feedback.


The disadvantages of the Iterative and Incremental SDLC Model are as follows –

- More resources may be required.
- Although cost of change is lesser, but it is not very suitable for changing requirements.
- More management attention is required.
- System architecture or design issues may arise because not all requirements are gathered in the beginning of the entire life cycle.
- Defining increments may require definition of the complete system.
- Not suitable for smaller projects.
- Management complexity is more.
- End of project may not be known which is a risk.
- Highly skilled resources are required for risk analysis.
- Projects progress is highly dependent upon the risk analysis phase.

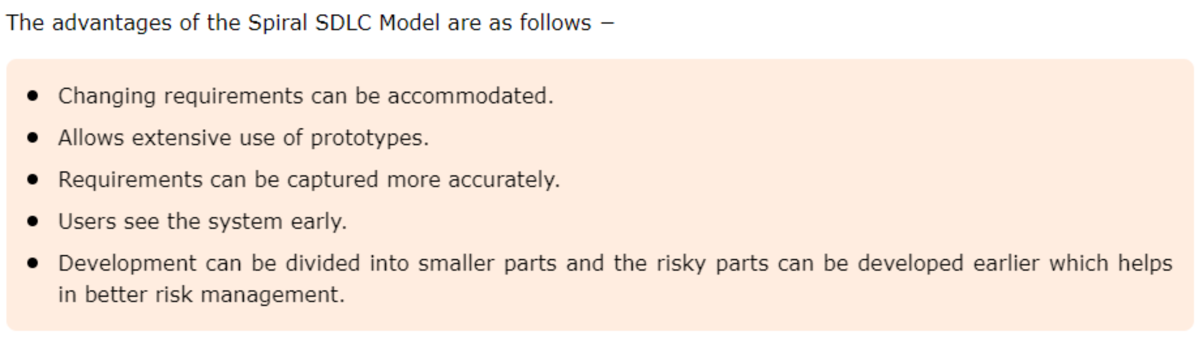
Spiral Model: It's easy to identify and manage risks, as requirements can change between iterations. However, repeated cycles could lead to scope change and underestimation of resources.

Identification: This phase starts with gathering the business requirements in the baseline spiral. In the subsequent spirals as the product matures, identification of system requirements, subsystem requirements and unit requirements are all done in this phase. This phase also includes understanding the system requirements by continuous communication between the customer and the system analyst. At the end of the spiral, the product is deployed in the identified market.

Design: The Design phase starts with the conceptual design in the baseline spiral and involves architectural design, logical design of modules, physical product design and the final design in the subsequent spirals.

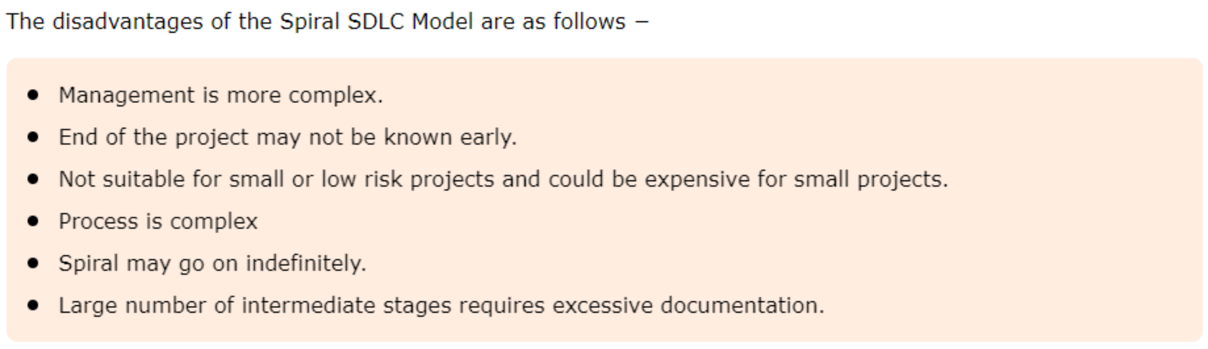
Construct or Build: The Construct phase refers to production of the actual software product at every spiral. In the baseline spiral, when the product is just thought of and the design is being developed a POC (Proof of Concept) is developed in this phase to get customer feedback.Then in the subsequent spirals with higher clarity on requirements and design details a working model of the software called build is produced with a version number. These builds are sent to the customer for feedback.

Evaluation and Risk Analysis: Risk Analysis includes identifying, estimating and monitoring the technical feasibility and management risks, such as schedule slippage and cost overrun. After testing the build, at the end of first iteration, the customer evaluates the software and provides feedback.



The advantages of the Spiral SDLC Model are as follows −

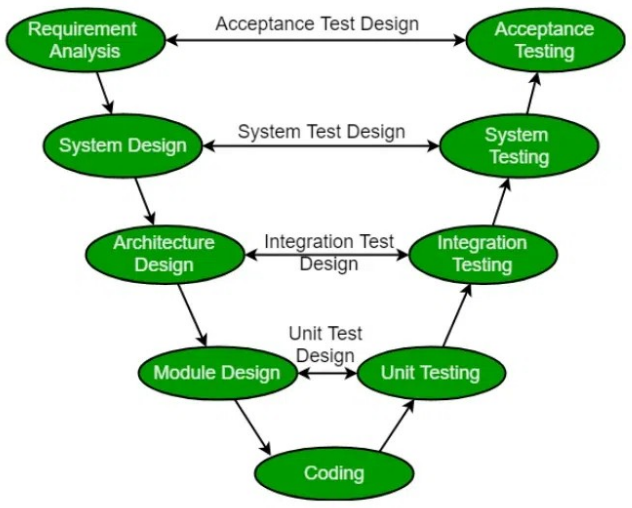
- Changing requirements can be accommodated.
- Allows extensive use of prototypes.
- Requirements can be captured more accurately.
- Users see the system early.
- Development can be divided into smaller parts and the risky parts can be developed earlier which helps in better risk management.



The disadvantages of the Spiral SDLC Model are as follows −

- Management is more complex.
- End of the project may not be known early.
- Not suitable for small or low risk projects and could be expensive for small projects.
- Process is complex
- Spiral may go on indefinitely.
- Large number of intermediate stages requires excessive documentation.

V-Model: It is also known as Verification and Validation model. Under the V-Model, the corresponding testing phase of the development phase is planned in parallel. So, there are Verification phases on one side of the 'V' and Validation phases on the other side. The Coding Phase joins the two sides of the V-Model.



V-Model - Verification Phases

There are several Verification phases in the V-Model, each of these are explained in detail below.

Business Requirement Analysis: This is the first phase in the development cycle where the product requirements are understood from the customer's perspective. This phase involves detailed communication with the customer to understand his expectations and exact requirement. This is a very important activity and needs to be managed well, as most of the customers are not sure about what exactly they need. The acceptance test design planning is done at this stage as business requirements can be used as an input for acceptance testing.

System Design: Once you have the clear and detailed product requirements, it is time to design the complete system. The system design will have the understanding and detailing the complete hardware and communication setup for the product under development. The system test plan is developed based on the system design. Doing this at an earlier stage leaves more time for the actual test execution later.

Architectural Design: Architectural specifications are understood and designed in this phase. Usually more than one technical approach is proposed and based on the technical and financial feasibility the final decision is taken. The system design is broken down further into modules taking up different functionality. This is also referred to as High Level Design (HLD). The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood and defined in this stage. With this information, integration tests can be designed and documented during this stage.

Module Design: In this phase, the detailed internal design for all the system modules is specified, referred to as Low Level Design (LLD). It is important that the design is compatible with the other modules in the system architecture and the other external systems. The unit tests are an essential part of any development process and helps eliminate the maximum faults and errors at a very early stage. These unit tests can be designed at this stage based on the internal module designs.

Coding Phase

The actual coding of the system modules designed in the design phase is taken up in the Coding phase. The best suitable programming language is decided based on the system and architectural requirements. The coding is performed based on the coding guidelines and standards. The code goes through numerous code reviews and is optimized for best performance before the final build is checked into the repository.

Validation Phases

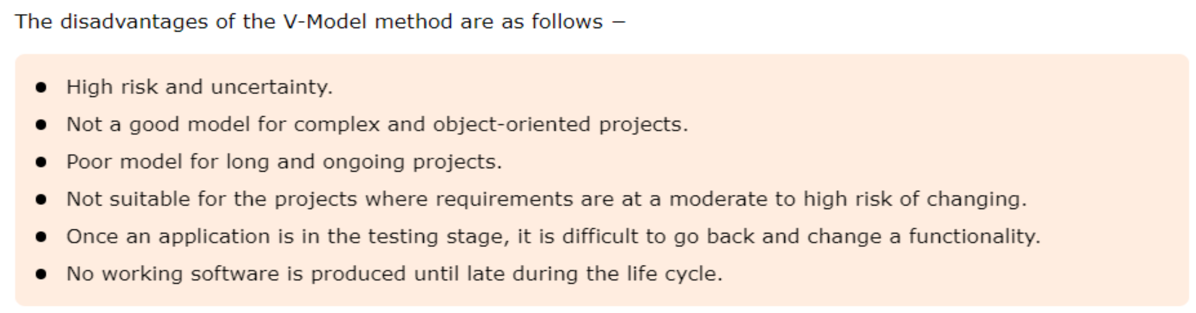
The different Validation Phases in a V-Model are explained in detail below.

Unit Testing: Unit tests designed in the module design phase are executed on the code during this validation phase. Unit testing is the testing at code level and helps eliminate bugs at an early stage, though all defects cannot be uncovered by unit testing.

Integration Testing: Integration testing is associated with the architectural design phase. Integration tests are performed to test the coexistence and communication of the internal modules within the system.

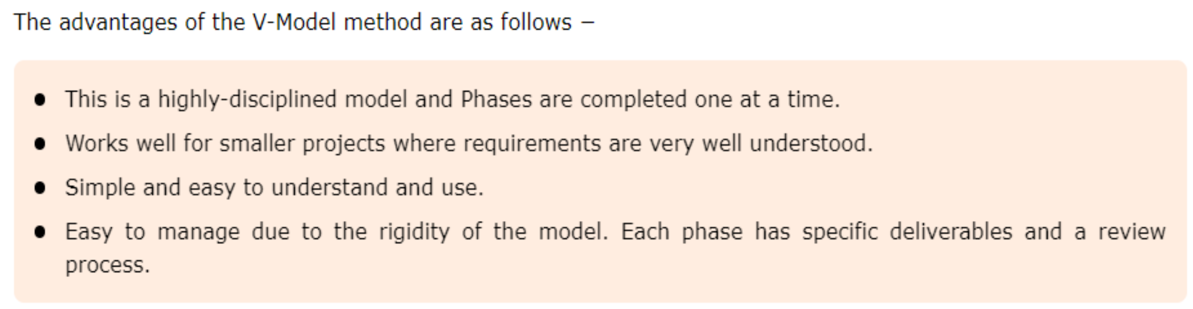
System Testing: System testing is directly associated with the system design phase. System tests check the entire system functionality and the communication of the system under development with external systems. Most of the software and hardware compatibility issues can be uncovered during this system test execution.

Acceptance Testing: Acceptance testing is associated with the business requirement analysis phase and involves testing the product in user environment. Acceptance tests uncover the compatibility issues with the other systems available in the user environment. It also discovers the non-functional issues such as load and performance defects in the actual user environment.



The disadvantages of the V-Model method are as follows −

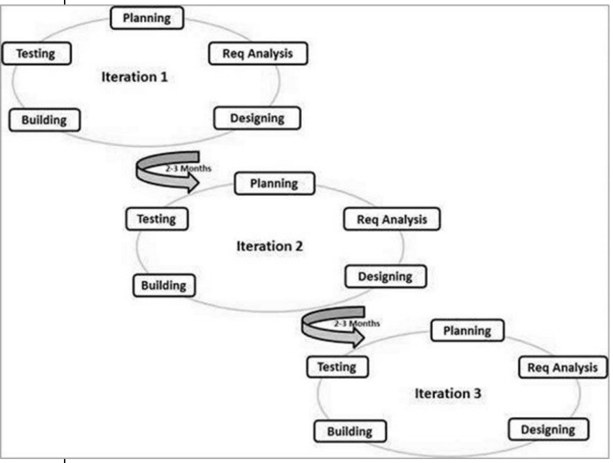
- High risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.
- Once an application is in the testing stage, it is difficult to go back and change a functionality.
- No working software is produced until late during the life cycle.

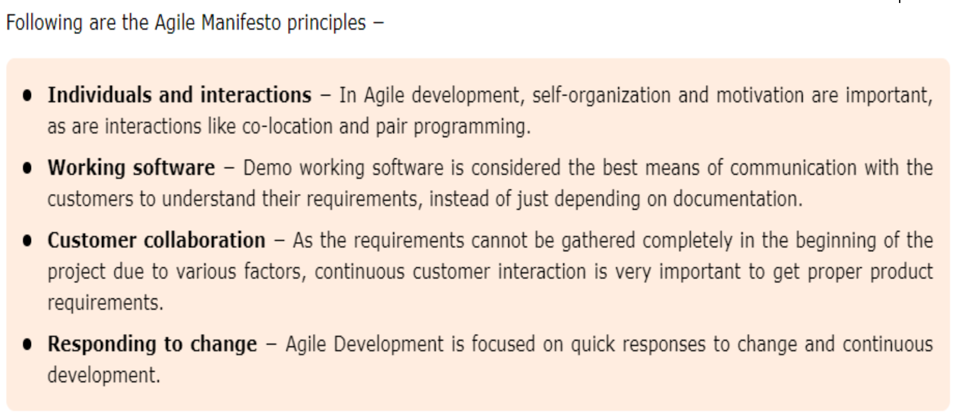


The advantages of the V-Model method are as follows −

- This is a highly-disciplined model and Phases are completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Simple and easy to understand and use.
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.

Agile Model: The agile model arranges the SDLC phases into several development cycles. The team iterates through the phases rapidly, delivering only small, incremental software changes in each cycle. They continuously evaluate requirements, plans, and results so that they can respond quickly to change. The agile model is both iterative and incremental, making it more efficient than other process models.
Rapid development cycles help teams identify and address issues in complex projects early on and before they become significant problems. They can also engage customers and stakeholders to obtain feedback throughout the project lifecycle. However, overreliance on customer feedback could lead to excessive scope changes or end the project midway.

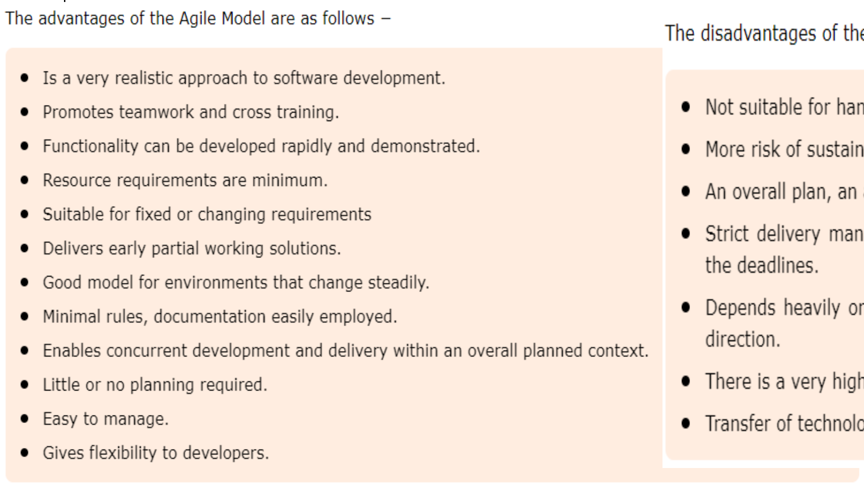



Following are the Agile Manifesto principles −

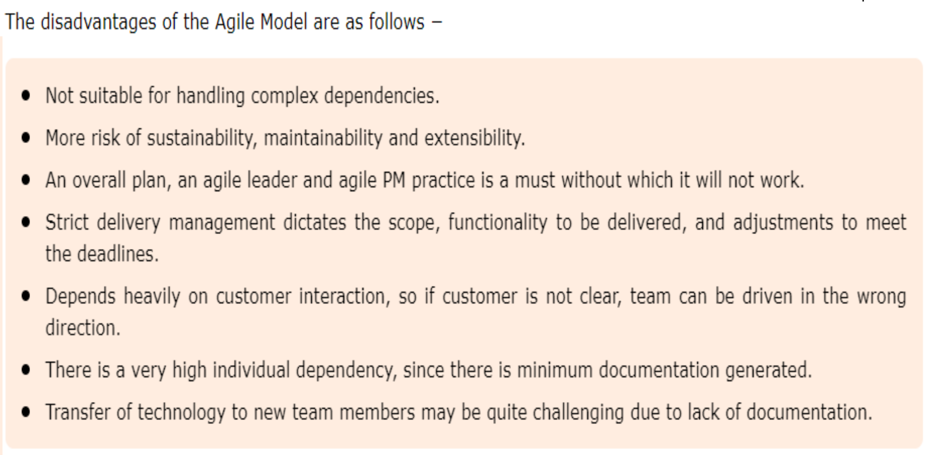
- **Individuals and interactions** − In Agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
- **Working software** − Demo working software is considered the best means of communication with the customers to understand their requirements, instead of just depending on documentation.
- **Customer collaboration** − As the requirements cannot be gathered completely in the beginning of the project due to various factors, continuous customer interaction is very important to get proper product requirements.
- **Responding to change** − Agile Development is focused on quick responses to change and continuous development.

Agile Vs Traditional SDLC Models

Agile is based on the adaptive software development methods, whereas the traditional SDLC models like the waterfall model is based on a predictive approach. Predictive teams in the traditional SDLC models usually work with detailed planning and have a complete forecast of the exact tasks and features to be delivered in the next few months or during the product life cycle. Predictive methods entirely depend on the requirement analysis and planning done in the beginning of cycle. Any changes to be incorporated go through a strict change control management and prioritization. Agile uses an adaptive approach where there is no detailed planning and there is clarity on future tasks only in respect of what features need to be developed. There is feature driven development and the team adapts to the changing product requirements dynamically. The product is tested very frequently, through the release iterations, minimizing the risk of any major failures in future. Customer Interaction is the backbone of this Agile methodology, and open communication with minimum documentation are the typical features of Agile development environment. The agile teams work in close collaboration with each other and are most often located in the same geographical location.


The advantages of the Agile Model are as follows −

- Is a very realistic approach to software development.
- Promotes teamwork and cross training.
- Functionality can be developed rapidly and demonstrated.
- Resource requirements are minimum.
- Suitable for fixed or changing requirements
- Delivers early partial working solutions.
- Good model for environments that change steadily.
- Minimal rules, documentation easily employed.
- Enables concurrent development and delivery within an overall planned context.
- Little or no planning required.
- Easy to manage.
- Gives flexibility to developers.


The disadvantages of the Agile Model are as follows −

- Not suitable for handling complex dependencies.
- More risk of sustainability, maintainability and extensibility.
- An overall plan, an agile leader and agile PM practice is a must without which it will not work.
- Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
- Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
- There is a very high individual dependency, since there is minimum documentation generated.
- Transfer of technology to new team members may be quite challenging due to lack of documentation.

Prototype: Software prototyping is becoming very popular as a software development model, as it enables to understand customer requirements at an early stage of development. It helps get valuable feedback from the customer and helps software designers and developers understand about what exactly is expected from the product under development.

Prototype is a working model of software with some limited functionality. The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation.

Prototyping is used to allow the users evaluate developer proposals and try them out before implementation. It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

Software Prototyping is most useful in development of systems having high level of user interactions such as online systems. Systems which need users to fill out forms or go through various screens before data is processed can use prototyping very effectively to give the exact look and feel even before the actual software is developed.

Software that involves too much of data processing and most of the functionality is internal with very little user interface does not usually benefit from prototyping. Prototype development could be an extra overhead in such projects and may need lot of extra efforts.

The advantages of the Prototyping Model are as follows −

- Increased user involvement in the product even before its implementation.
- Since a working model of the system is displayed, the users get a better understanding of the system being developed.
- Reduces time and cost as the defects can be detected much earlier.
- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily.
- Confusing or difficult functions can be identified.

The Disadvantages of the Prototyping Model are as follows −

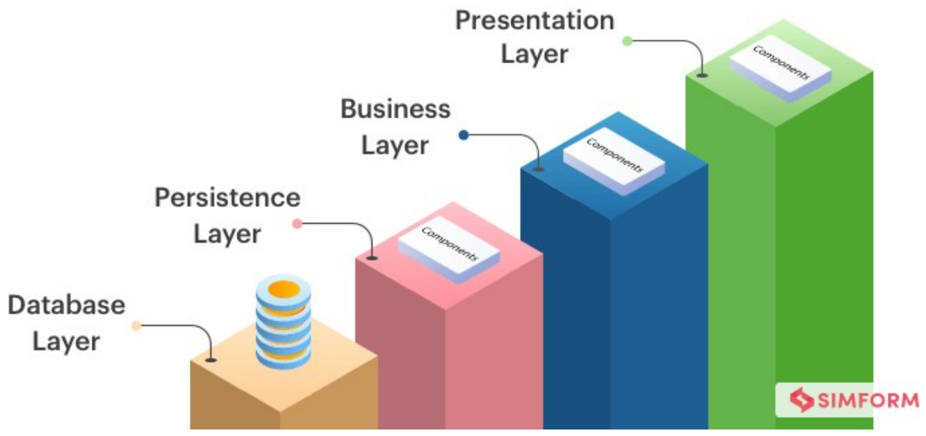
- Risk of insufficient requirement analysis owing to too much dependency on the prototype.
- Users may get confused in the prototypes and actual systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Developers may try to reuse the existing prototypes to build the actual system, even when it is not technically feasible.
- The effort invested in building prototypes may be too much if it is not monitored properly.

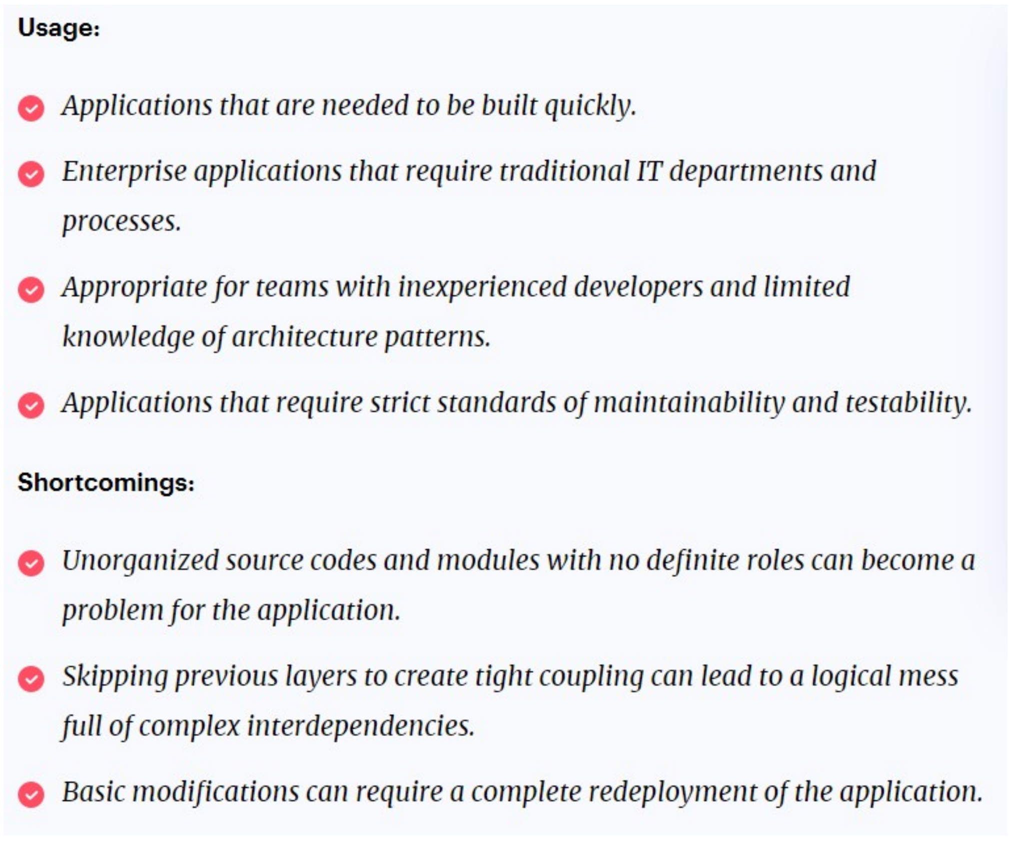
ARCHITECTURAL PATTERNS

Layered Architecture Pattern: Often, a layered architecture is classified into four distinct layers: presentation, business, persistence, and database; however, the pattern is not confined to the specified layers and there can be an application layer or service layer or data access layer.

This pattern stands out because each layer plays a distinct role within the application and is marked as closed. It means a request must pass through the layer below it to go to the next layer. Another one of its concepts – layers of isolation – enables you to modify components within one layer without affecting the other layers.

Let's take an example of an eCommerce web application. The business logic required to process a shopping cart activity, such as calculating the cart, is directly fetched from the application tier to the presentation tier. Here the application tier acts as an integration layer to establish seamless communication between the data and presentation layers. Additionally, the last tier is the data tier used to maintain data independently without the intervention of the application server and the business logic.





Usage:

- *Applications that are needed to be built quickly.*
- *Enterprise applications that require traditional IT departments and processes.*
- *Appropriate for teams with inexperienced developers and limited knowledge of architecture patterns.*
- *Applications that require strict standards of maintainability and testability.*

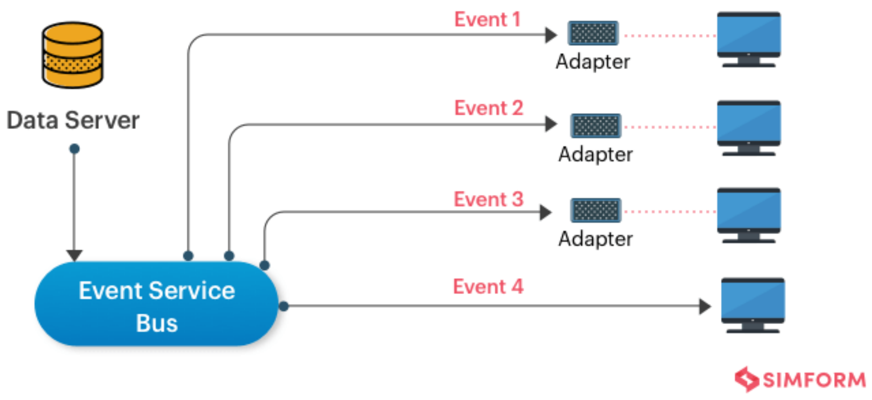
Shortcomings:

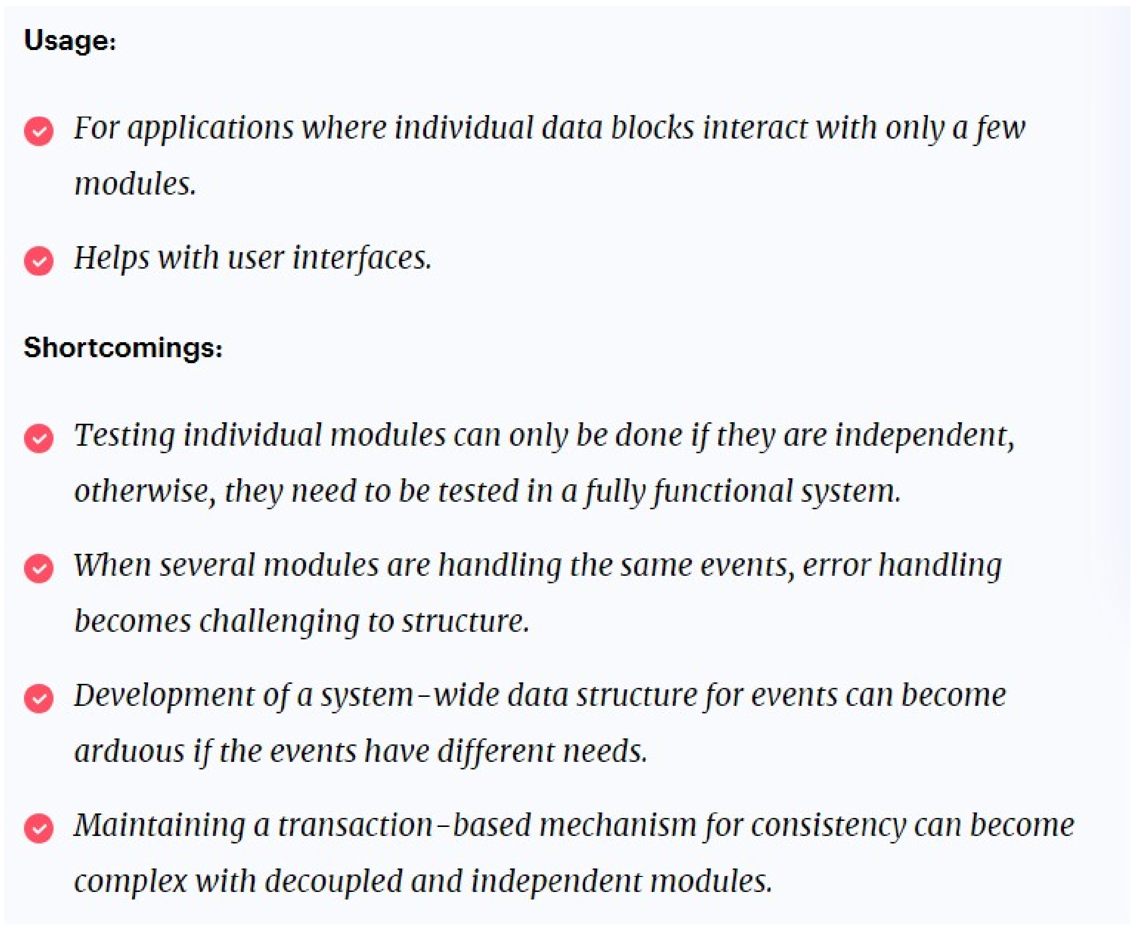
- *Unorganized source codes and modules with no definite roles can become a problem for the application.*
- *Skipping previous layers to create tight coupling can lead to a logical mess full of complex interdependencies.*
- *Basic modifications can require a complete redeployment of the application.*

Event-driven Architecture Pattern: It is made up of decoupled, single-purpose event processing components that asynchronously receive and process events. This pattern orchestrates the behavior around the production, detection, and consumption of all the events, along with the responses they evoke.

It is made up of decoupled, single-purpose event processing components that asynchronously receive and process events. This pattern orchestrates the behavior around the production, detection, and consumption of all the events, along with the responses they evoke.

A good example that uses event-driven architecture is an e-commerce site. The event-driven architecture enables the e-commerce website to react to various sources at a time of high demand. Simultaneously, it avoids any crash of the application or any over-provisioning of resources.





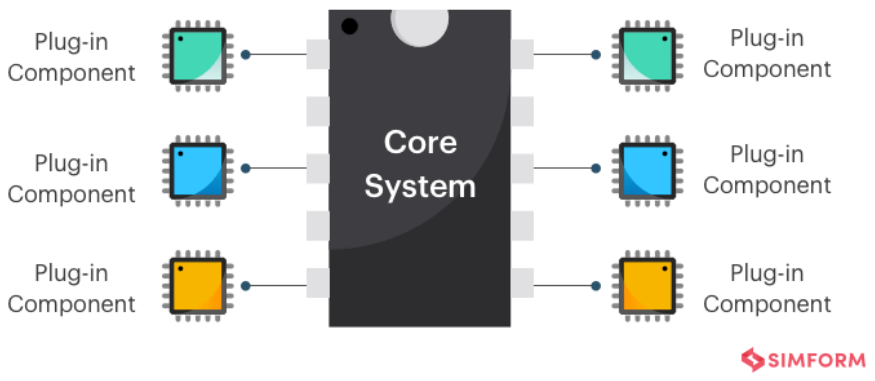
Usage:

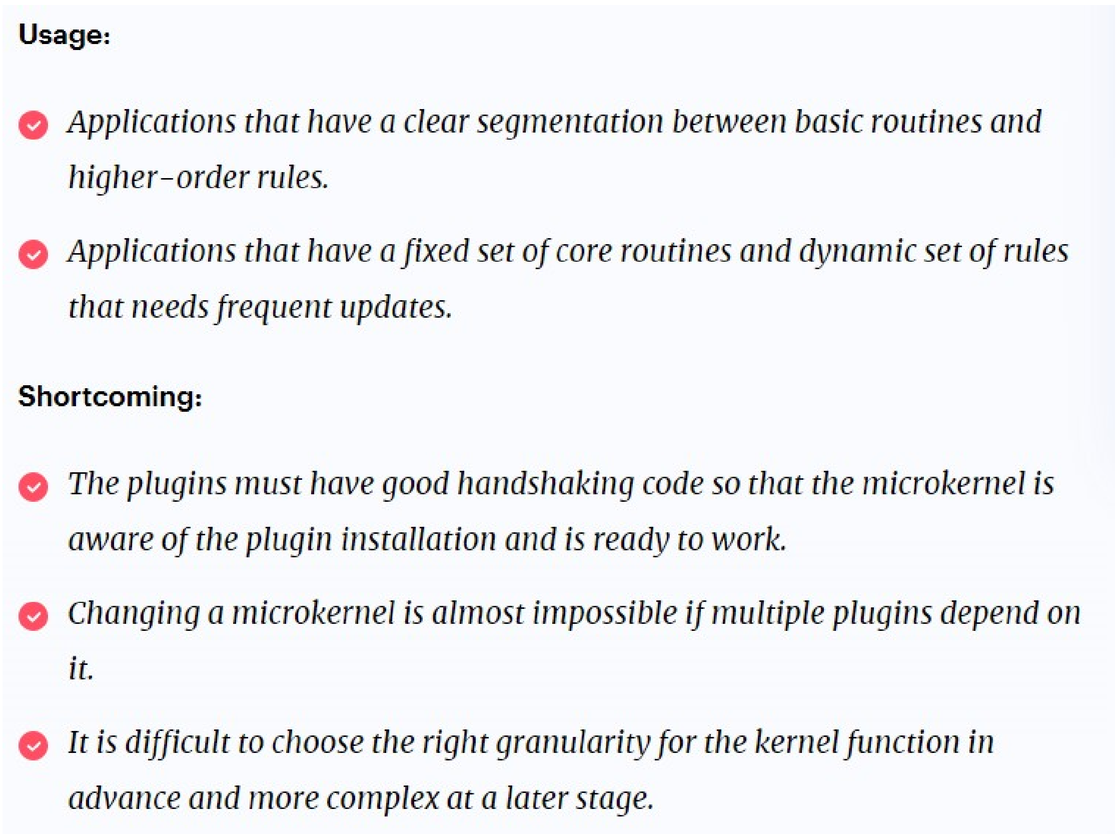
- *For applications where individual data blocks interact with only a few modules.*
- *Helps with user interfaces.*

Shortcomings:

- *Testing individual modules can only be done if they are independent, otherwise, they need to be tested in a fully functional system.*
- *When several modules are handling the same events, error handling becomes challenging to structure.*
- *Development of a system-wide data structure for events can become arduous if the events have different needs.*
- *Maintaining a transaction-based mechanism for consistency can become complex with decoupled and independent modules.*

Microkernel Architecture Pattern: This architecture pattern consists of two types of components – a core system and several plug-in modules. While the core system works on minimal functionality to keep the system operational, the plug-in modules are independent components with specialized processing.
If we take a business application's perspective, the core system can be defined as general business logic without the custom code for special cases, special rules, or complex conditional processes. On the other hand, the plug-in modules are meant to enhance the core system in order to produce additional business capabilities.





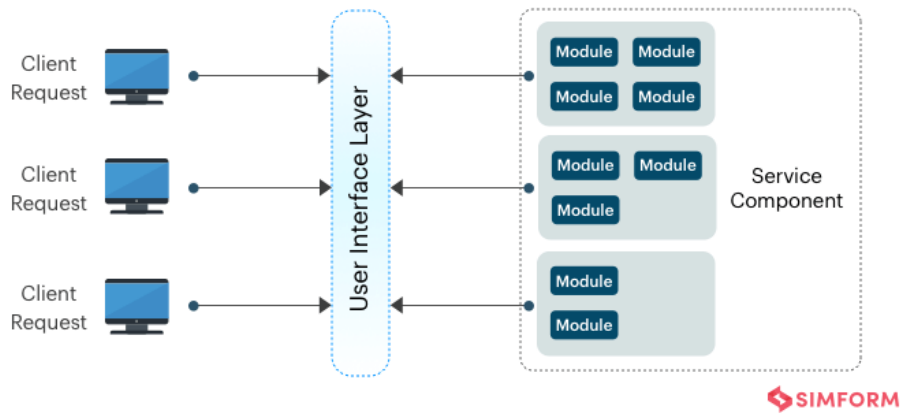
Usage:

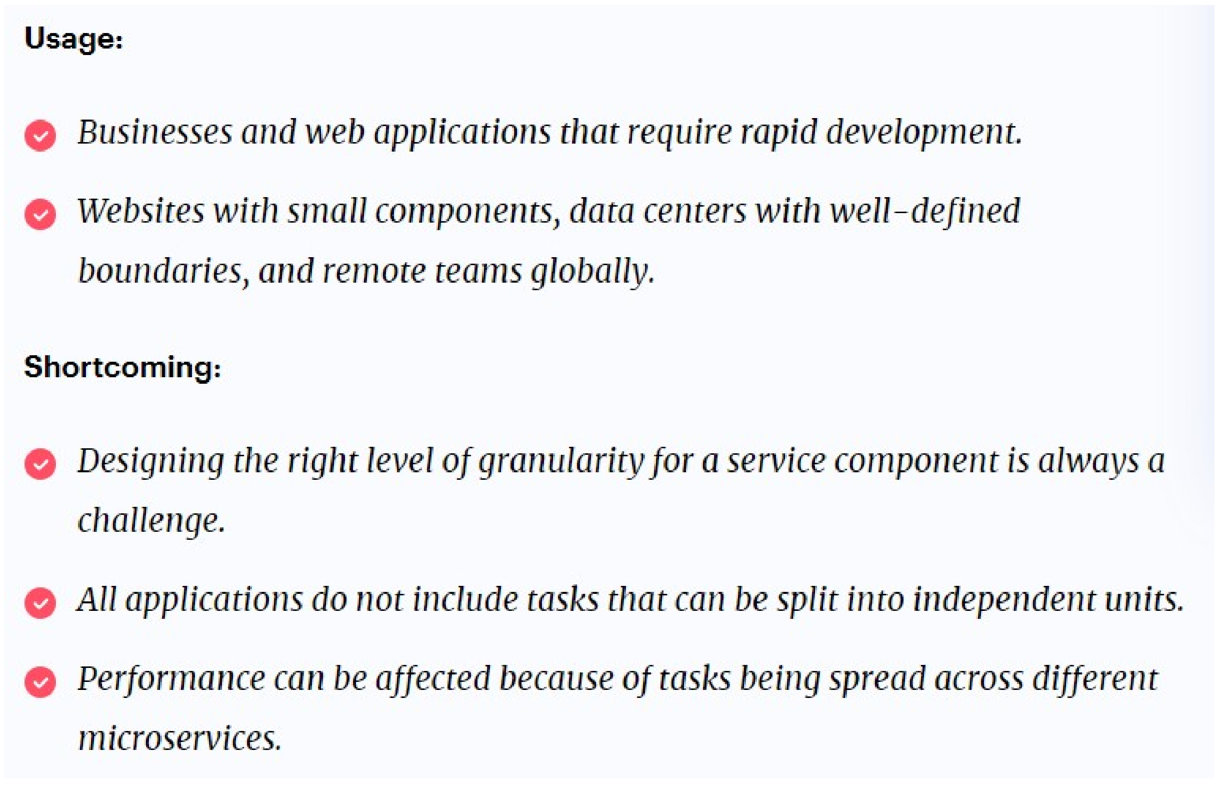
- *Applications that have a clear segmentation between basic routines and higher-order rules.*
- *Applications that have a fixed set of core routines and dynamic set of rules that needs frequent updates.*

Shortcoming:

- *The plugins must have good handshaking code so that the microkernel is aware of the plugin installation and is ready to work.*
- *Changing a microkernel is almost impossible if multiple plugins depend on it.*
- *It is difficult to choose the right granularity for the kernel function in advance and more complex at a later stage.*

Microservices Architecture Pattern: Microservices architecture pattern is seen as a viable alternative to monolithic applications and service-oriented architectures. The components are deployed as separate units through an effective, streamlined delivery pipeline. The pattern's benefits are enhanced scalability and a high degree of decoupling within the application.
Owing to its decoupled and independent characteristics, the components are accessed through a remote access protocol. Moreover, the same components can be separately developed, deployed, and tested without interdependency on any other service component.
Netflix is one of the early adopters of the microservice architecture pattern. The architecture allowed the engineering team to work in small teams responsible for the end-to-end development of hundreds of microservices. These microservices work together to stream digital entertainment to millions of Netflix customers every day.





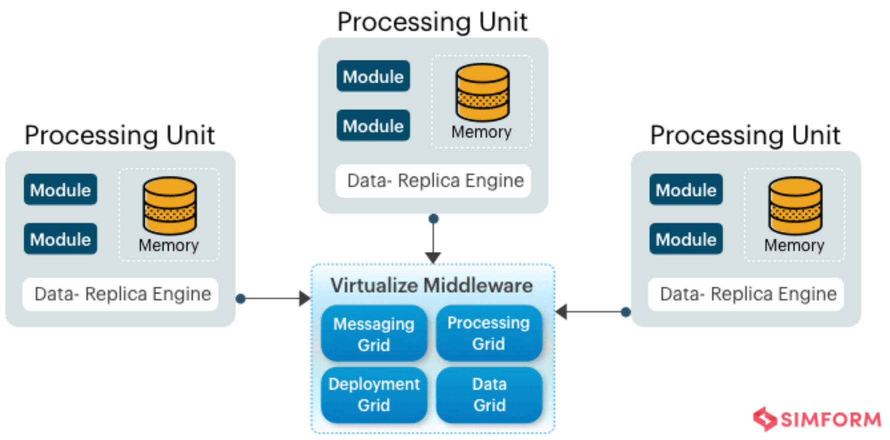
Usage:

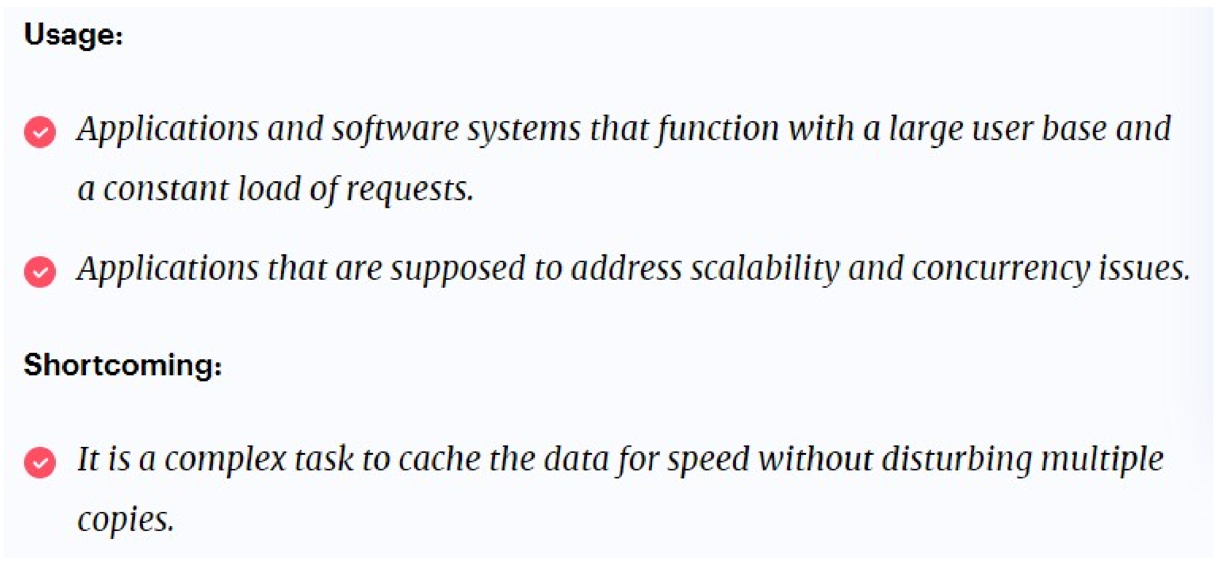
- *Businesses and web applications that require rapid development.*
- *Websites with small components, data centers with well-defined boundaries, and remote teams globally.*

Shortcoming:

- *Designing the right level of granularity for a service component is always a challenge.*
- *All applications do not include tasks that can be split into independent units.*
- *Performance can be affected because of tasks being spread across different microservices.*

Space-Based Architecture Pattern: The concept of tuple space – the idea of distributed shared memory is the basis of the name of this architecture. The space-based pattern comprises two primary components – a processing unit and a virtualized middleware.
The processing unit contains portions of application components, including web-based components and backend business logic. While smaller web applications could be deployed in a single processing unit, the larger applications could split the application functionality into multiple processing units to avoid functional collapse. Furthermore, the virtualized-middleware component contains elements that control various aspects of data synchronization and request handling. They can be custom-written or can be purchased as third-party products. SBA is indeed designed to distribute processing power across multiple nodes in a distributed system. It's particularly well-suited for scenarios where you need to process large volumes of data in real-time, like real-time analytics or high-performance transaction processing.



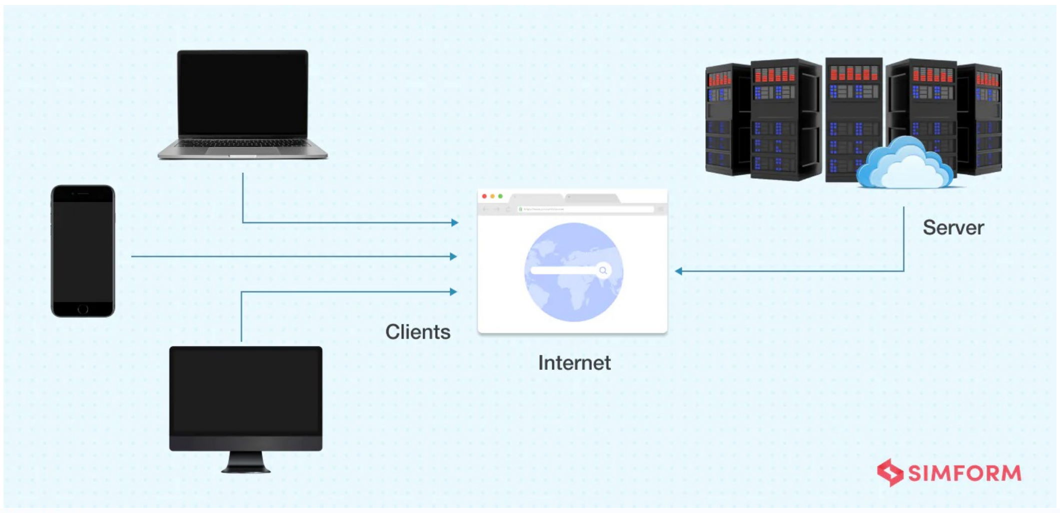

Usage:

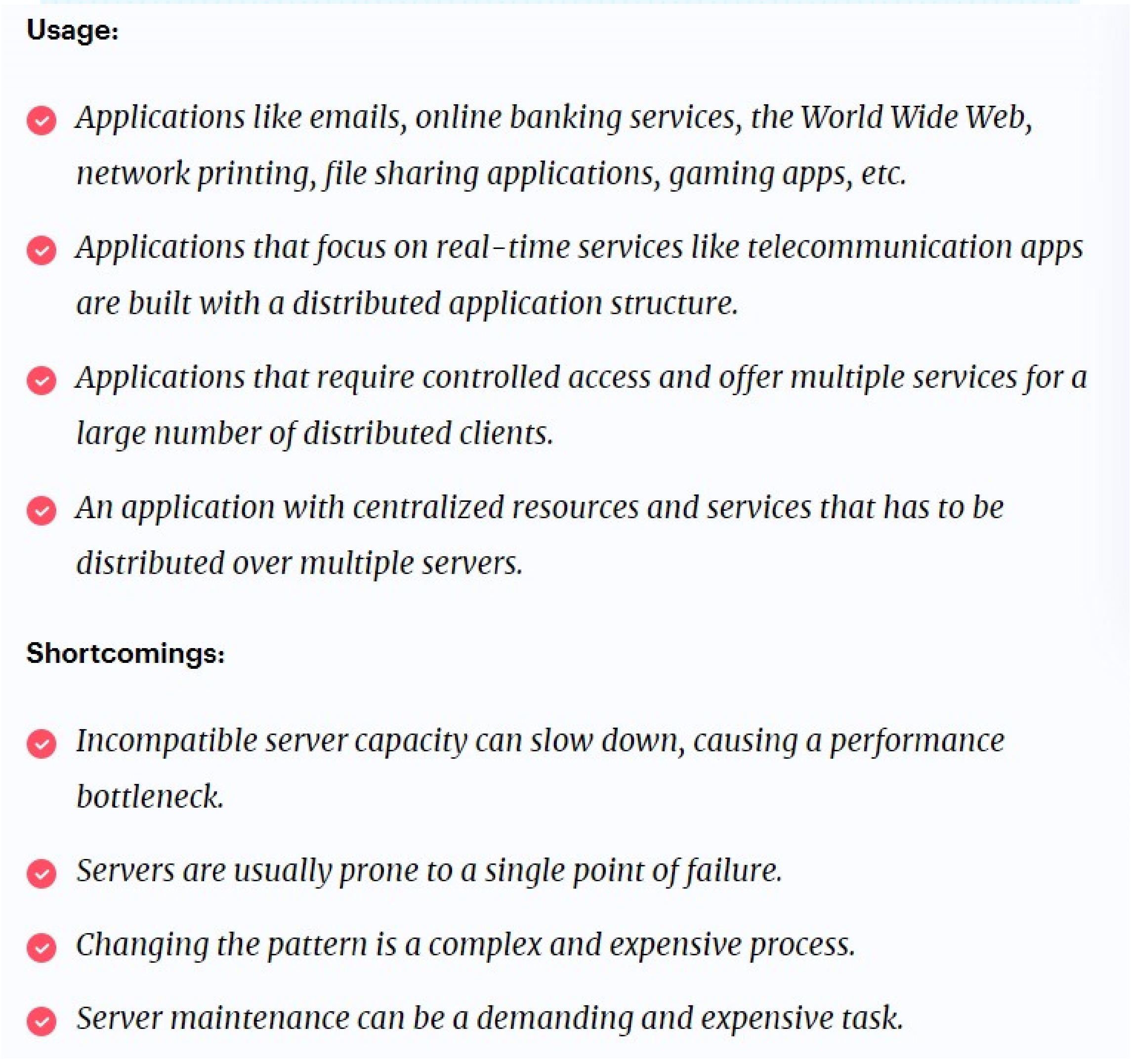
- *Applications and software systems that function with a large user base and a constant load of requests.*
- *Applications that are supposed to address scalability and concurrency issues.*

Shortcoming:

- *It is a complex task to cache the data for speed without disturbing multiple copies.*

Client-Server Architecture Pattern: A client-server architecture pattern is described as a distributed application structure having two main components – a client and a server. This architecture facilitates the communication between the client and the server, which may or may not be under the same network. A client requests specific resources to be fetched from the server, which might be in the form of data, content, services, files, etc. The server identifies the requests made and responds to the client appropriately by sending over the requested resources.





Usage:

- *Applications like emails, online banking services, the World Wide Web, network printing, file sharing applications, gaming apps, etc.*
- *Applications that focus on real-time services like telecommunication apps are built with a distributed application structure.*
- *Applications that require controlled access and offer multiple services for a large number of distributed clients.*
- *An application with centralized resources and services that has to be distributed over multiple servers.*

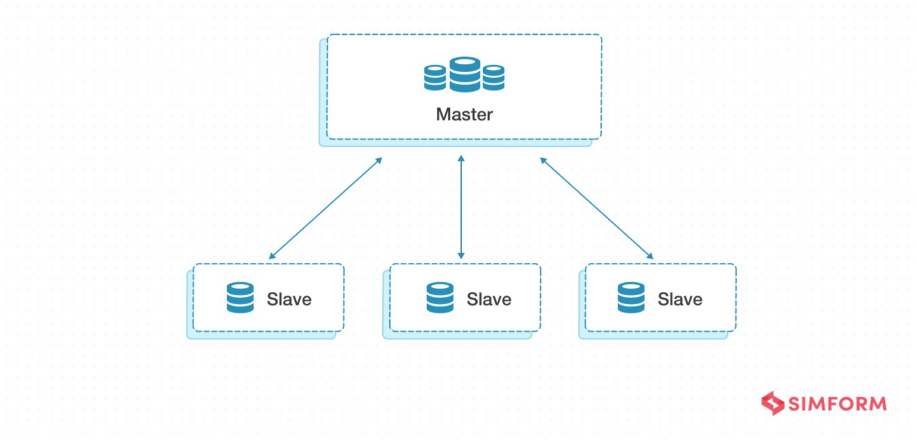
Shortcomings:

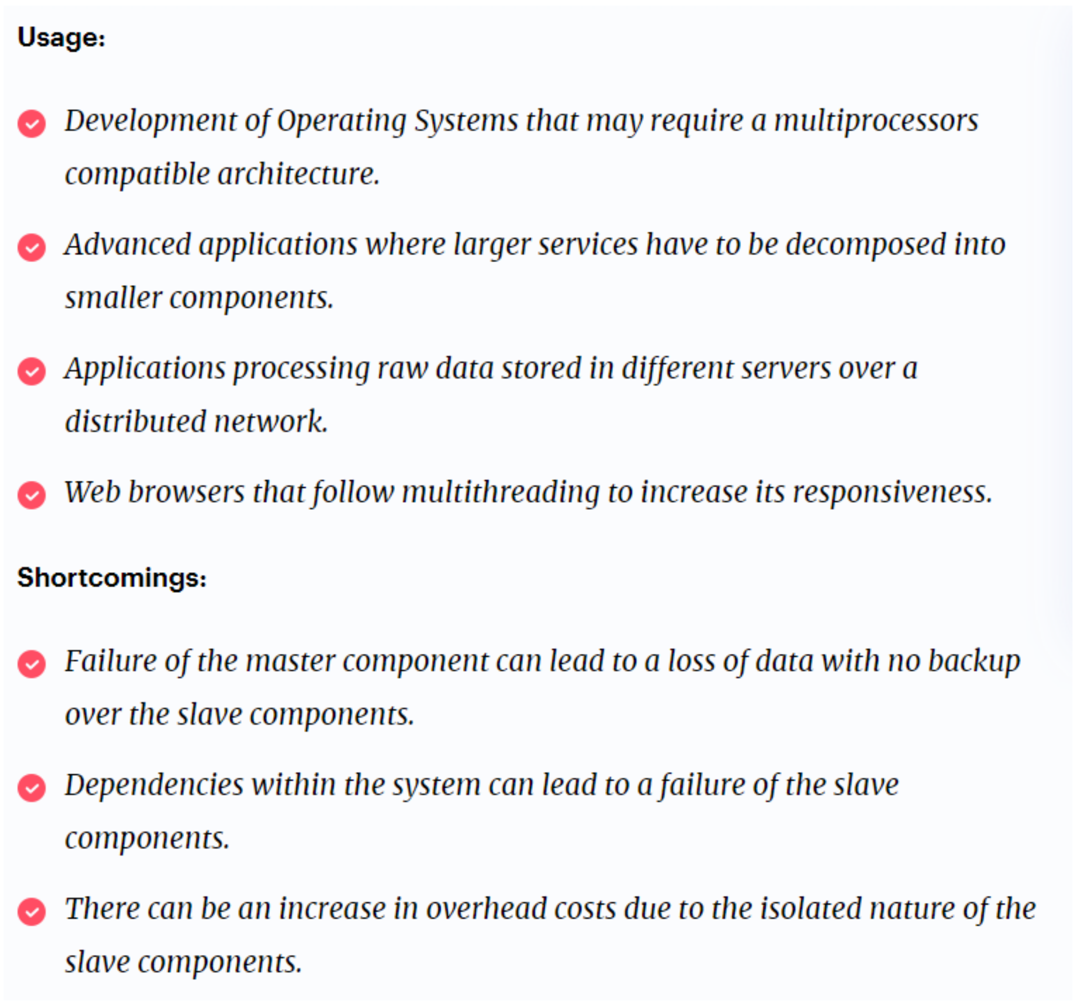
- *Incompatible server capacity can slow down, causing a performance bottleneck.*
- *Servers are usually prone to a single point of failure.*
- *Changing the pattern is a complex and expensive process.*
- *Server maintenance can be a demanding and expensive task.*

Master-Slave Architecture Pattern: Imagine a single database receiving multiple similar requests at the same time. Naturally, processing every single request at the same time can complicate and slow down the application process. A solution to this problem is a master-slave architecture pattern that functions with the master database launching multiple slave components to process those requests quickly.

As the title suggests, the master-slave architecture pattern can be pictured as a master distributing tasks to its slaves. Once the slave components finish their tasks, the distributed tasks are compiled by the master and displayed as the result.

As the title suggests, the master-slave architecture pattern can be pictured as a master distributing tasks to its slaves. Once the slave components finish their tasks, the distributed tasks are compiled by the master and displayed as the result.





Usage:

- *Development of Operating Systems that may require a multiprocessors compatible architecture.*
- *Advanced applications where larger services have to be decomposed into smaller components.*
- *Applications processing raw data stored in different servers over a distributed network.*
- *Web browsers that follow multithreading to increase its responsiveness.*

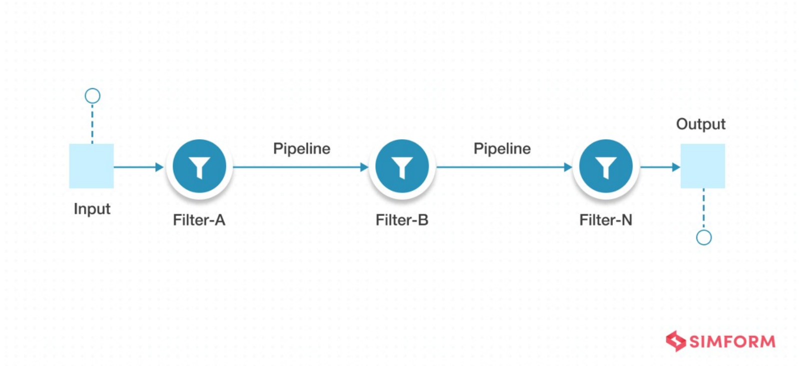
Shortcomings:

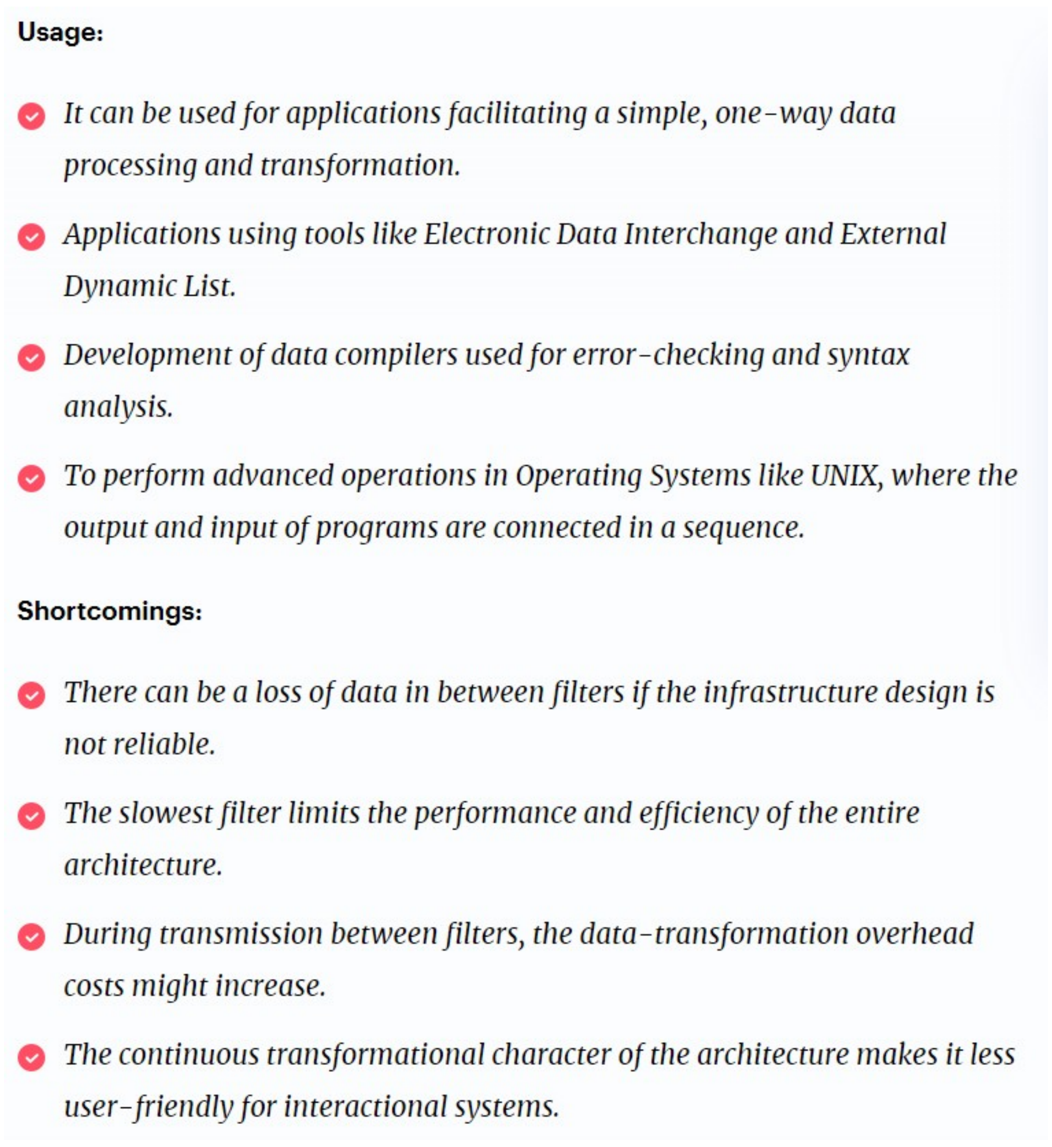
- *Failure of the master component can lead to a loss of data with no backup over the slave components.*
- *Dependencies within the system can lead to a failure of the slave components.*
- *There can be an increase in overhead costs due to the isolated nature of the slave components.*

Pipe-Filter Architecture Pattern: A pipe-filter architecture pattern processes a stream of data in a unidirectional flow where components are referred to as filters, and pipes are those which connect these filters. The chain of processing data takes place where the pipes transmit data to the filters, and the result of one filter becomes the input for the next filter. The function of this architecture is to break down significant components/processes into independent and multiple components that can be processed simultaneously.

The pipe-filter pattern is best suited for applications that process data in a stream using web services and can create simple sequences to complex structures. Compilers can be considered a fitting example having this architecture pattern since each filter performs lexical analysis, parsing, semantic analysis, and code generation.

Example: AAA servers





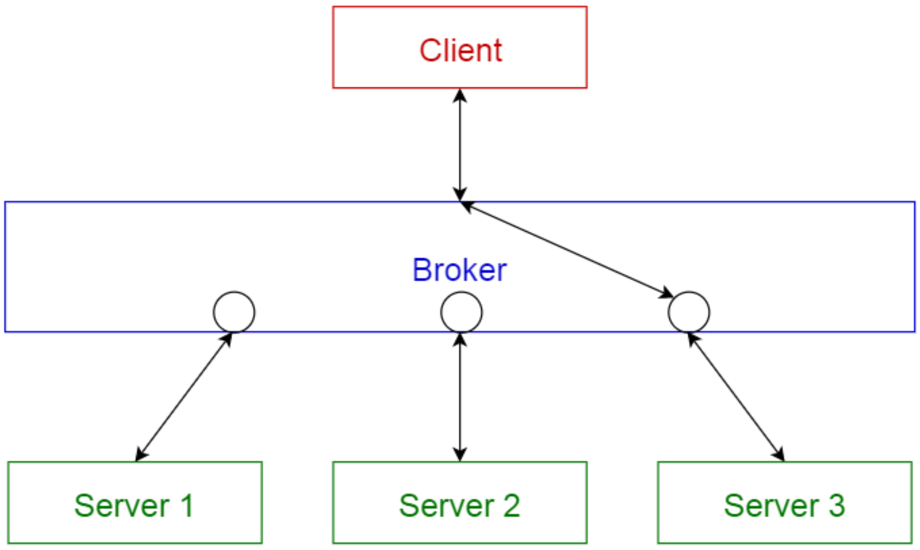
Usage:

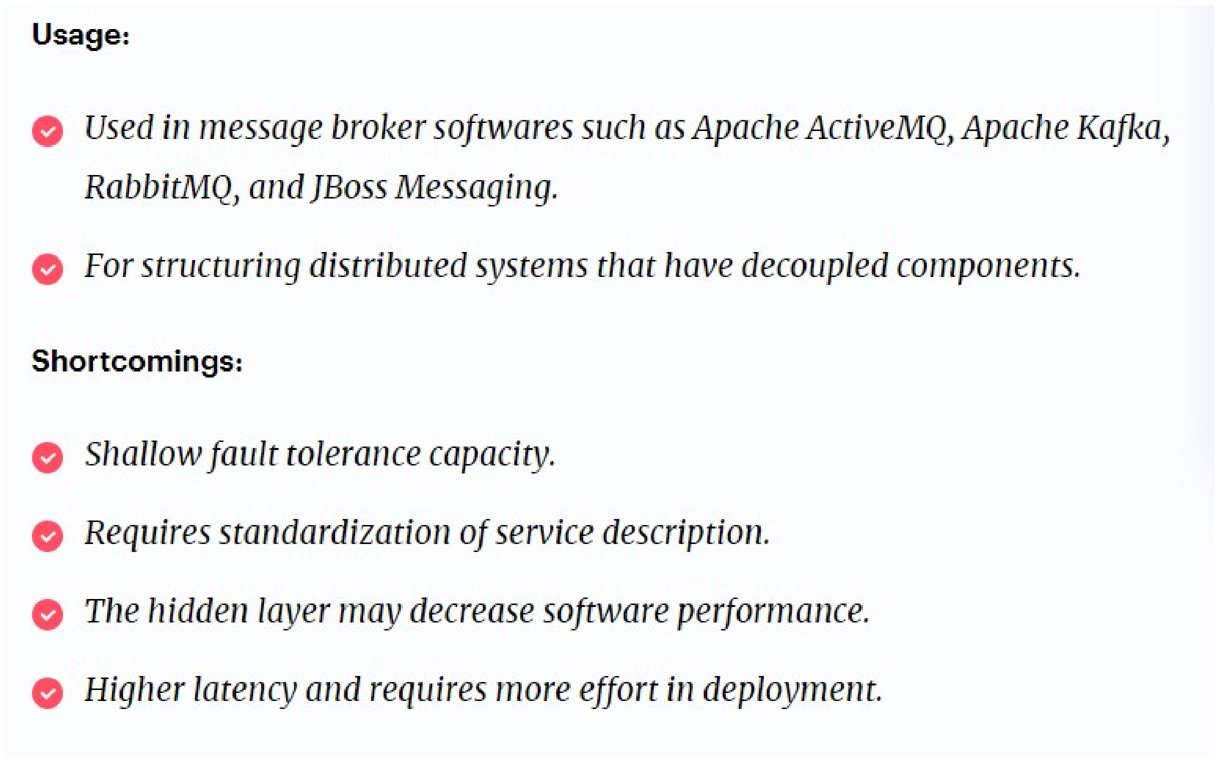
- *It can be used for applications facilitating a simple, one-way data processing and transformation.*
- *Applications using tools like Electronic Data Interchange and External Dynamic List.*
- *Development of data compilers used for error-checking and syntax analysis.*
- *To perform advanced operations in Operating Systems like UNIX, where the output and input of programs are connected in a sequence.*

Shortcomings:

- *There can be a loss of data in between filters if the infrastructure design is not reliable.*
- *The slowest filter limits the performance and efficiency of the entire architecture.*
- *During transmission between filters, the data-transformation overhead costs might increase.*
- *The continuous transformational character of the architecture makes it less user-friendly for interactional systems.*

Broker Architecture Pattern: A broker pattern is used for structuring distributed systems with decoupled components. By invoking remote services, components can interact with others in broker architecture patterns. Also, the broker is responsible for all the coordination and communication among the components.
Clients, servers, and brokers are three major components of the broker pattern. Generally, a broker will have access to all the services and characteristics related to a particular server. When clients request a service from the broker, the broker redirects them to a suitable service category for further process.
It is simply a Load Balancer. The Broker pattern acts as a mediator between different components in a distributed system, handling message routing, queuing, and sometimes transformation. It doesn't inherently distribute processing power; instead, it helps to decouple communication between components, which can improve scalability and maintainability.





Usage:

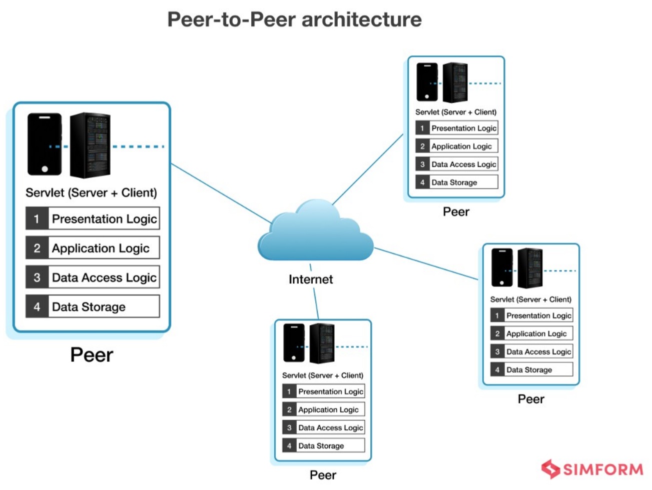
- *Used in message broker softwares such as Apache ActiveMQ, Apache Kafka, RabbitMQ, and JBoss Messaging.*
- *For structuring distributed systems that have decoupled components.*

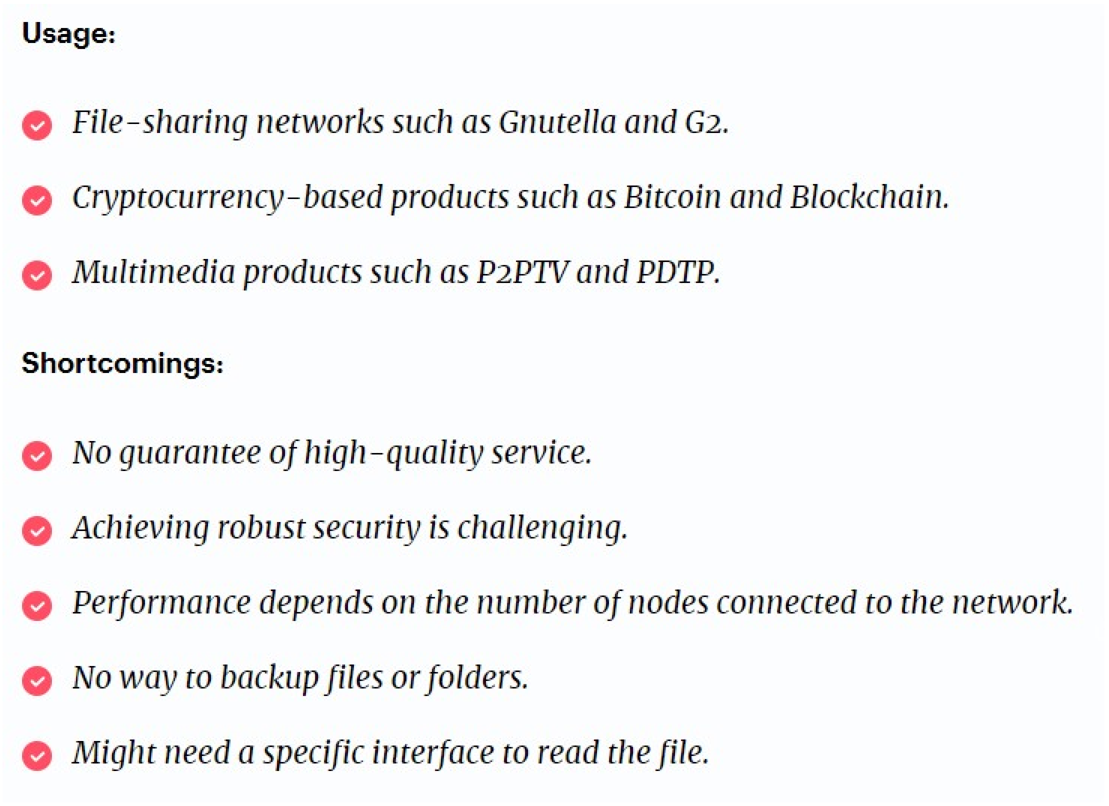
Shortcomings:

- *Shallow fault tolerance capacity.*
- *Requires standardization of service description.*
- *The hidden layer may decrease software performance.*
- *Higher latency and requires more effort in deployment.*

Peer-to-Peer Architecture Pattern: In the peer-to-peer architectural pattern, individual components are called peers. A peer can act as a client, a server, or both and change its role dynamically over time. As a client, a peer can request service from other peers, and as a server, a peer can provide services to other peers. The significant difference between peer-to-peer and client-server architecture is that each computer on the network has considerable authority and the absence of a centralized server. Its capacity increases as more and more computers join the network.

An excellent example of a peer-to-peer architecture pattern would be file-sharing networks like Skype, BitTorrent, and Napster. In BitTorrent, peer-to-peer architecture is used for distributing the data and files on the internet in a decentralized fashion. By using this protocol, one can transfer large video and audio files with the utmost ease. In Skype, you use the VoIP P2P architecture pattern to make a voice call and send text messages to another user. In this manner, you can use peer-to-peer architecture for file sharing, messaging, collaboration, etc.





Usage:

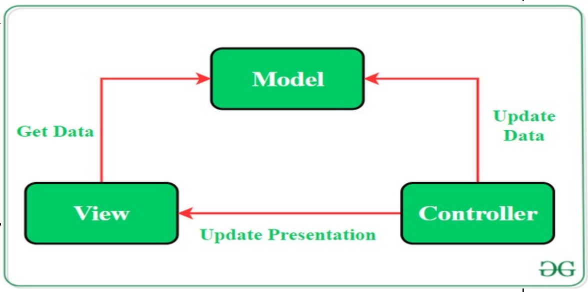
- *File-sharing networks such as Gnutella and G2.*
- *Cryptocurrency-based products such as Bitcoin and Blockchain.*
- *Multimedia products such as P2PTV and PDTP.*

Shortcomings:

- *No guarantee of high-quality service.*
- *Achieving robust security is challenging.*
- *Performance depends on the number of nodes connected to the network.*
- *No way to backup files or folders.*
- *Might need a specific interface to read the file.*

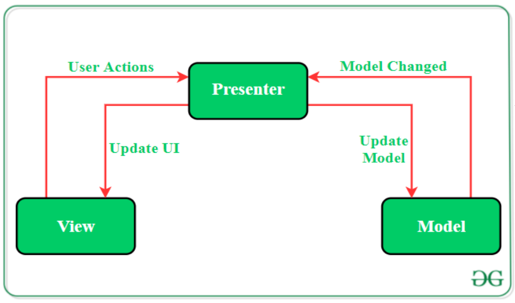
ANDROID PATTERNS

The Model—View—Controller(MVC) Pattern: MVC pattern is the oldest android app architecture which simply suggests separating the code into 3 different layers:



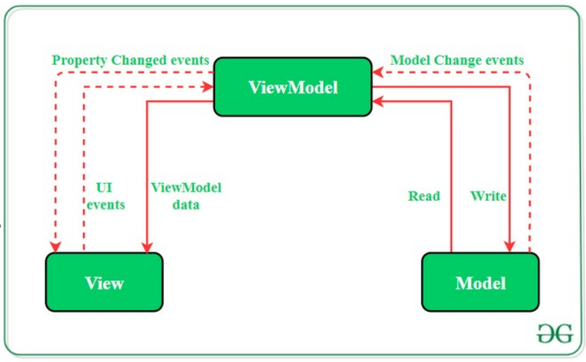
Model: Layer for storing data. It is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.
View: UI(User Interface) layer. It provides the visualization of the data stored in the Model.
Controller: Layer which contains core logic. It gets informed of the user's behavior and updates the Model as per the need.
Advantages: MVC pattern increases the code testability and makes it easier to implement new features as it highly supports the separation of concerns. Unit testing of the Model and Controller is possible as they do not extend or use any Android class. Functionalities of the View can be checked through UI tests if the View respect the single responsibility principle(update controller and display data from the model without implementing domain logic)
Disadvantages: Code layers depend on each other even if MVC is applied correctly. No parameter to handle UI logic i.e., how to display the data.

The Model—View—Presenter(MVP) Pattern: MVP pattern is the second iteration of Android app architecture. This pattern is widely accepted and is still recommended for upcoming developers. The purpose of each component is easy to learn:



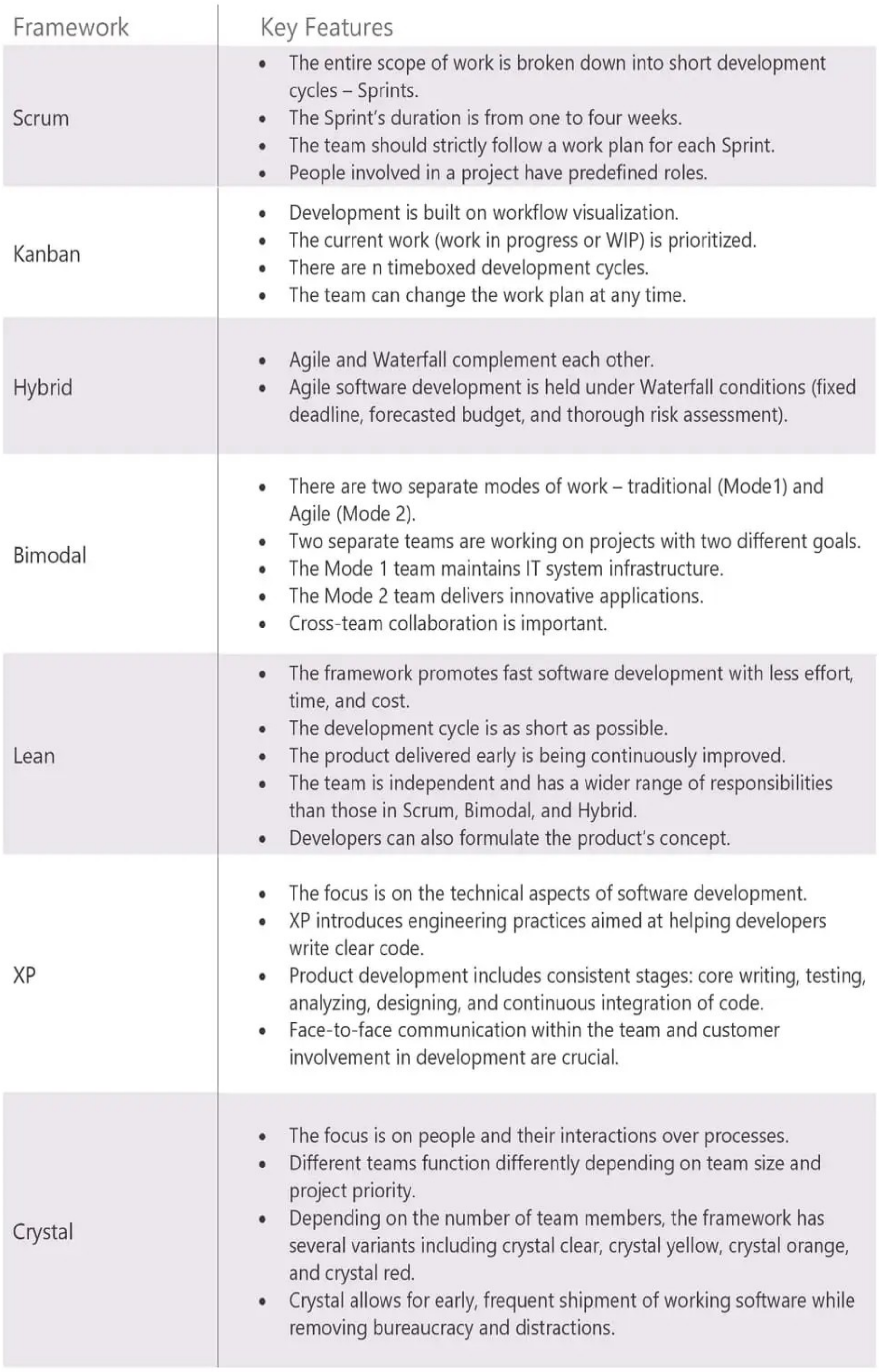
Model: Layer for storing data. It is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.
View: UI(User Interface) layer. It provides the visualization of the data and keep a track of the user's action in order to notify the Presenter.
Presenter: Fetch the data from the model and applies the UI logic to decide what to display. It manages the state of the View and takes actions according to the user's input notification from the View.
Advantages: No conceptual relationship in android components. Easy code maintenance and testing as the application's model, view, and presenter layer are separated.
Disadvantages: If the developer does not follow the single responsibility principle to break the code then the Presenter layer tends to expand to a huge all-knowing class.

The Model—View—ViewModel (MVVM) Pattern: While releasing the Android Architecture Components, the Android team recommended this architecture pattern. Below are the separate code layers:
Model: This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.



View: The purpose of this layer is to inform the ViewModel about the user's action.
ViewModel: It exposes those data streams which are relevant to the View.
The MVVM and MVP patterns are quite similar because both are efficient in abstracting the state and behavior of the View layer. In MVVM, Views can bind itself to the data streams which are exposed by ViewModel.

AGILE PROJECT MANAGEMENT



Framework	Key Features
Scrum	• The entire scope of work is broken down into short development cycles – Sprints. • The Sprint's duration is from one to four weeks. • The team should strictly follow a work plan for each Sprint. • People involved in a project have predefined roles.
Kanban	• Development is built on workflow visualization. • The current work (work in progress or WIP) is prioritized. • There are n timeboxed development cycles. • The team can change the work plan at any time.
Hybrid	• Agile and Waterfall complement each other. • Agile software development is held under Waterfall conditions (fixed deadline, forecasted budget, and thorough risk assessment).
Bimodal	• There are two separate modes of work – traditional (Mode1) and Agile (Mode 2). • Two separate teams are working on projects with two different goals. • The Mode 1 team maintains IT system infrastructure. • The Mode 2 team delivers innovative applications. • Cross-team collaboration is important.
Lean	• The framework promotes fast software development with less effort, time, and cost. • The development cycle is as short as possible. • The product delivered early is being continuously improved. • The team is independent and has a wider range of responsibilities than those in Scrum, Bimodal, and Hybrid. • Developers can also formulate the product's concept.
XP	• The focus is on the technical aspects of software development. • XP introduces engineering practices aimed at helping developers write clear code. • Product development includes consistent stages: core writing, testing, analyzing, designing, and continuous integration of code. • Face-to-face communication within the team and customer involvement in development are crucial.
Crystal	• The focus is on people and their interactions over processes. • Different teams function differently depending on team size and project priority. • Depending on the number of team members, the framework has several variants including crystal clear, crystal yellow, crystal orange, and crystal red. • Crystal allows for early, frequent shipment of working software while removing bureaucracy and distractions.

The Agile approach is often mistakenly considered to be a single methodology. Yet, there are dozens of methodologies and certain practices that have not been touched upon in this research.
Regardless of the exact methodologies and techniques they use, Agile teams have proven to increase profits 37 percent faster and generate 30 percent more revenue than non-Agile companies. Higher speed, flexibility, and productivity achieved through such approaches are the key drivers that motivate more and more organizations to switch to Agile.

Software engineering, being an extremely fast-paced industry, calls for flexibility and responsiveness in every aspect of project development. Agile methodologies allow for delivering cutting-edge products and cultivating innovative experiences while keeping the product in sync with market trends and user requirements.
However, there is always a place for diversity. Depending on your business requirements and goals, you might still benefit from using the Waterfall model or the combination of the two.

Agile methodology implementation steps: Once you have decided that the Agile methodology fills the bill in terms of your company and projects, you need to know how to implement it successfully. While the process may differ from one company to another, there are a few general steps to follow.

Step 1: Get your manager and stakeholder buy-in

Before you start implementing the new way of doing projects, make sure that everyone is on the same page and supports the change. So, it’s a good idea to talk to the key players and get their buy-in by explaining the benefits of Agile, addressing any of their concerns, and answering questions.

Step 2: Start small

Since incremental progress is the cornerstone of Agile, it makes sense to begin the methodology implementation with one small project, evaluate the feedback, and then apply it to other projects within your organization.

Step 3: Get your team excited

The success of Agile projects relies on the ability of different team members to cooperate and communicate. If your team isn’t excited about the whole idea and/or doesn’t support change, implementing Agile will be a tough task. After all, one of the main principles of Agile puts individuals and their interactions over processes and tools.

Step 4: Choose a fitting framework and stick to it

As you see, there are a lot of different Agile frameworks and practices to use. Each one has different requirements and focus. It’s important to pick an Agile framework that fits your processes in the best way and stick to it. If, say, you decide to implement Scrum, make sure your team strictly follows a work plan for each Sprint and attends daily meetings.

When to use Scrum: Scrum works well for long-term, complex projects that require stakeholder feedback, which may greatly affect project requirements. So, when the exact amount of work can’t be estimated, and the release date is not fixed, Scrum may be the best choice. The list of companies using this approach is impressive. In fact, there is a public spreadsheet with such organizations, including Microsoft, IBM, Yahoo, and Google. Scrum goes beyond IT. Companies working in the fields of finance, consulting, education, retail, media, and entertainment choose this approach to organize their work processes and enhance cooperation with customers.

When to use Kanban: Using Kanban, teams can do small releases and adapt to changing priorities. Unlike Scrum, there are no Sprints with their predefined goals. Kanban is focused on doing small pieces of work as they come up. For example, if testers find errors in the product, developers try to fix them right away. Kanban, for instance, works well after the main release of the product and suits update and maintenance purposes. Companies like Spotify and Wooga (a leading mobile games development company) have been using this approach successfully over the years. Yet, 27 percent of organizations combine Scrum with Kanban techniques, using so-called Scrumban rather than the original frameworks.

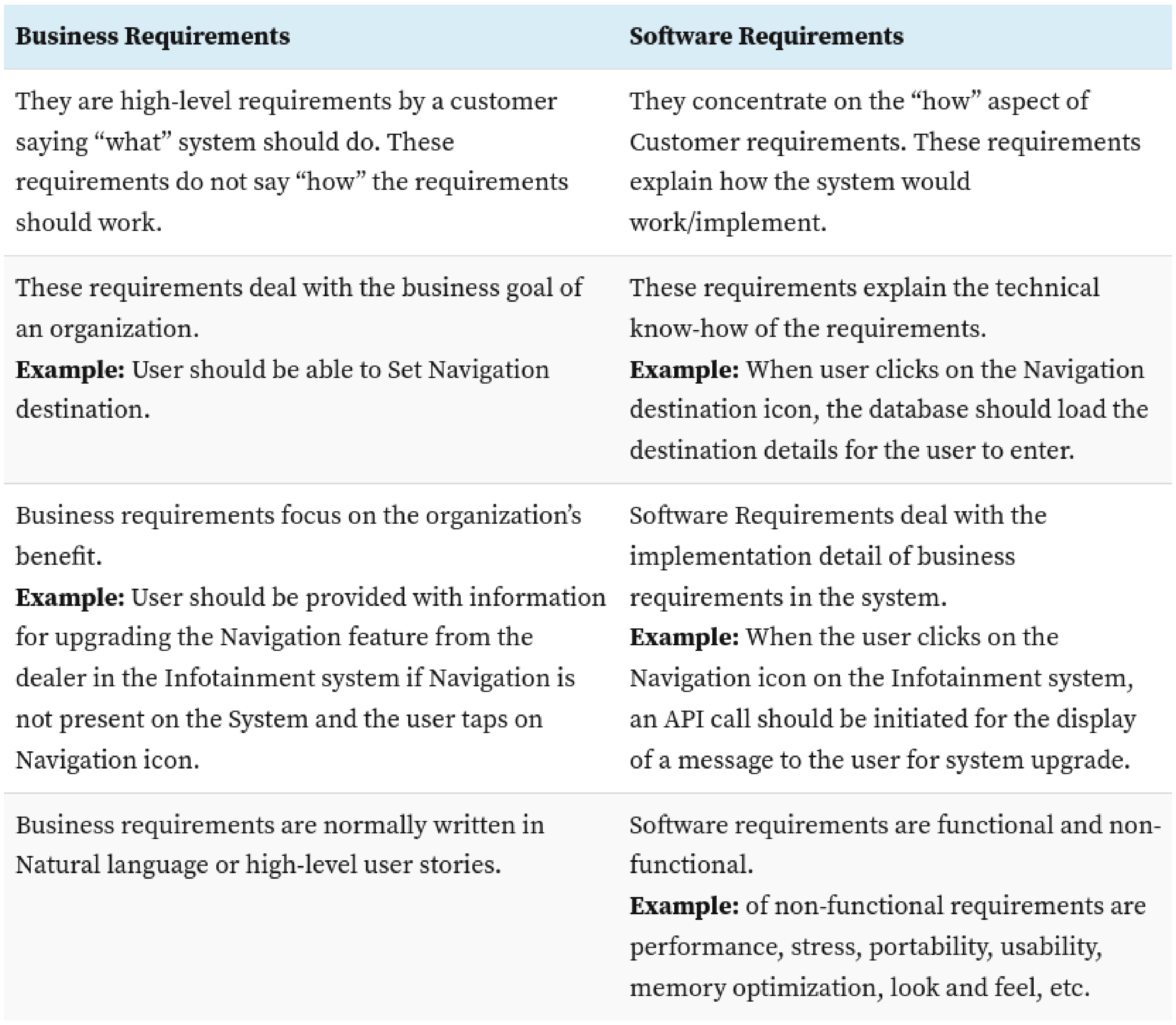
When to use Hybrid: Hybrid is an effective solution when product development includes both hardware and software. But, there is another reason to choose Hybrid. A situation in which a customer is not satisfied with an unspecified timeframe and budget, as well as a lack of planning, is not rare. Such uncertainty is typical for Agile. In this case, planning, requirements specification, and an application design can be accomplished in Waterfall while Agile is in place for software development and testing.

Requirement Analysis & User Interface for Target Audience

A good product delivery requires correct requirement gathering, the efficient examination of requirements that are gathered, and finally clear requirement documentation, as the precondition. This entire process is also called *Requirement Analysis in Software Development Life Cycle (SDLC).*

Requirement Analysis Stages/Steps:

- ***Requirement gathering*** is also called elicitation.
- This is followed by ***analyzing*** the collected requirements to understand the correctness and feasibility of converting these requirements into a possible product.
- And finally, ***documenting*** the requirements collected

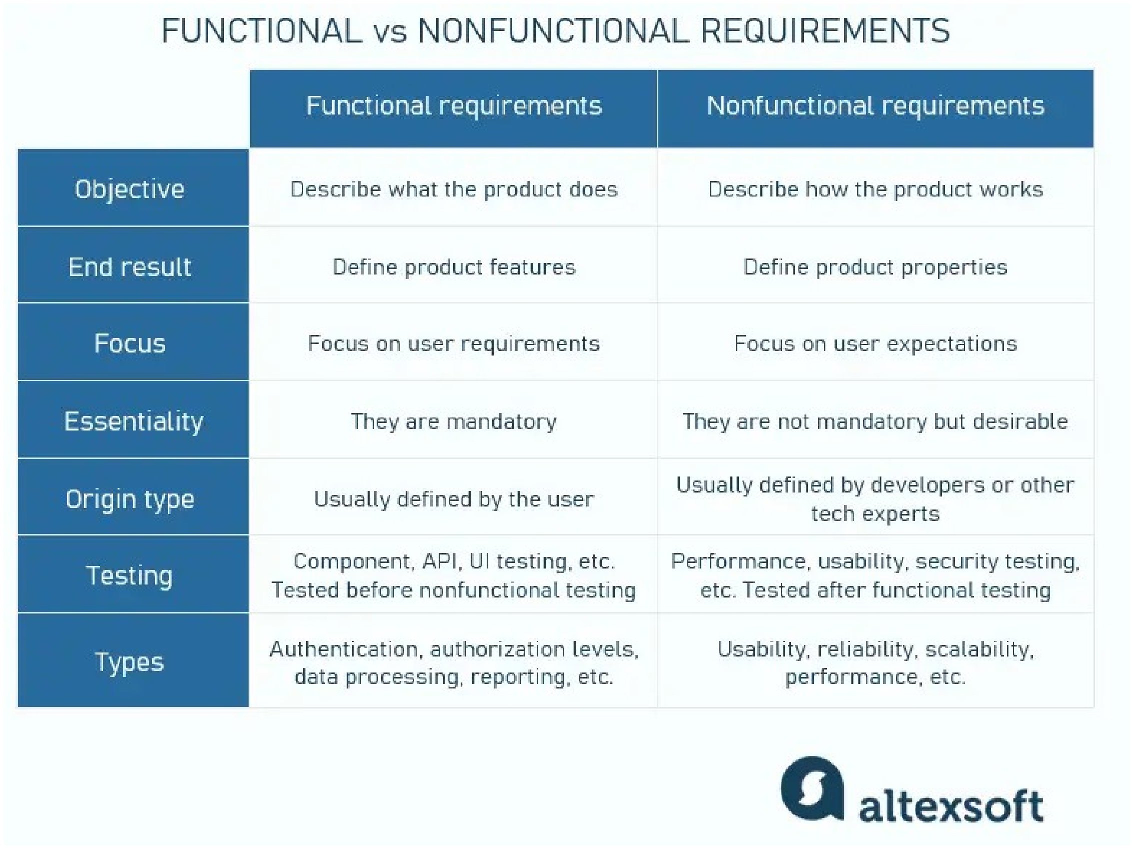


Business Requirements	Software Requirements
They are high-level requirements by a customer saying "what" system should do. These requirements do not say "how" the requirements should work.	They concentrate on the "how" aspect of Customer requirements. These requirements explain how the system would work/implement.
These requirements deal with the business goal of an organization. **Example:** User should be able to Set Navigation destination.	These requirements explain the technical know-how of the requirements. **Example:** When user clicks on the Navigation destination icon, the database should load the destination details for the user to enter.
Business requirements focus on the organization's benefit. **Example:** User should be provided with information for upgrading the Navigation feature from the dealer in the Infotainment system if Navigation is not present on the System and the user taps on Navigation icon.	Software Requirements deal with the implementation detail of business requirements in the system. **Example:** When the user clicks on the Navigation icon on the Infotainment system, an API call should be initiated for the display of a message to the user for system upgrade.
Business requirements are normally written in Natural language or high-level user stories.	Software requirements are functional and non-functional. **Example:** of non-functional requirements are performance, stress, portability, usability, memory optimization, look and feel, etc.

Functional and Non-Functional Requirements:

Functional requirements are product features or functions that developers must implement to enable users to accomplish their tasks. So it's essential to make them clear both for the development team and the stakeholders. Generally, functional requirements describe system behavior under specific conditions.

Nonfunctional requirements are not related to the system's functionality but rather define how the system should perform. They are crucial for ensuring the system's usability, reliability, and efficiency, often influencing the overall user experience. We'll describe the main categories of nonfunctional requirements in detail further on.



FUNCTIONAL vs NONFUNCTIONAL REQUIREMENTS

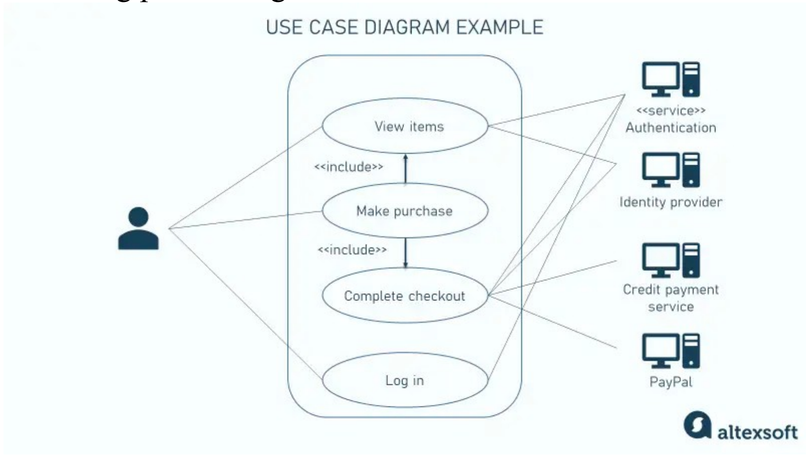
	Functional requirements	Nonfunctional requirements
Objective	Describe what the product does	Describe how the product works
End result	Define product features	Define product properties
Focus	Focus on user requirements	Focus on user expectations
Essentiality	They are mandatory	They are not mandatory but desirable
Origin type	Usually defined by the user	Usually defined by developers or other tech experts
Testing	Component, API, UI testing, etc. Tested before nonfunctional testing	Performance, usability, security testing, etc. Tested after functional testing
Types	Authentication, authorization levels, data processing, reporting, etc.	Usability, reliability, scalability, performance, etc.

altexsoft

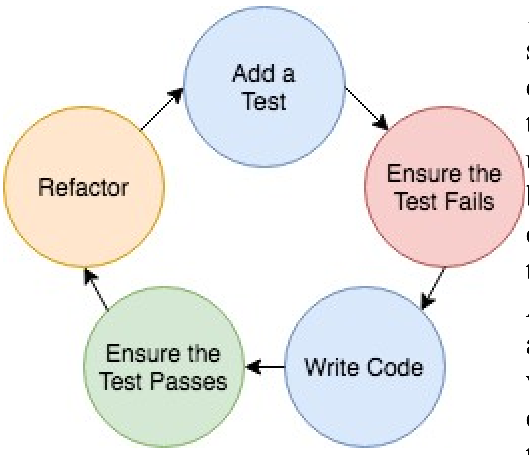
Requirement Formats:

a ***USER STORY*** is a documented description of a software functionality seen from the end-user perspective. The user story describes what exactly the user wants the system to do. In Agile projects, user stories are organized in a *backlog*. Currently, user stories are considered the best format for backlog items. *Example:* As an admin, I want to add product descriptions so that users can later view these descriptions and compare the products.

USE CASE describe the interaction between the system and external users that leads to achieving particular goals.



Test Driven Development - Behavior Driven Development



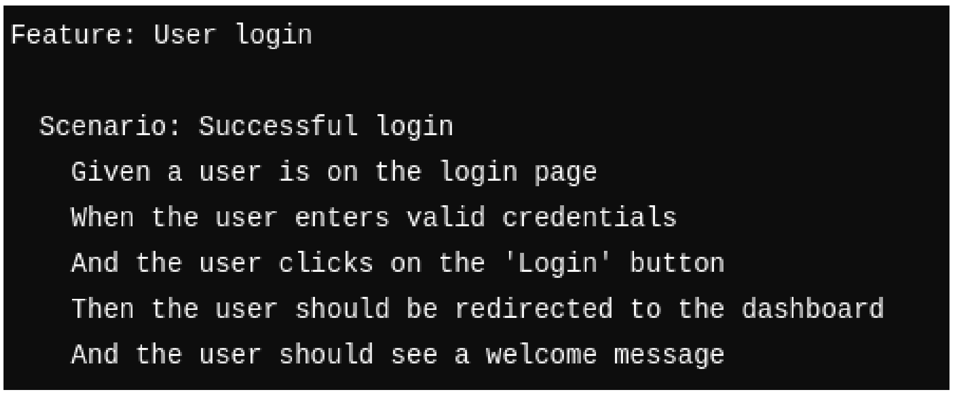
Test-Driven Development (TDD) is a methodology in software development that focuses on an iterative development cycle where the emphasis is placed on writing test cases *before* the actual feature or function is written. TDD utilizes repetition of short development cycles. It combines building and testing. This process not only helps ensure correctness of the code -- but also helps to indirectly evolve the design and architecture of the project at hand.
Alternative approaches to writing automated tests is to write all of the production code before starting on the test code or to write all of the test code before starting on the production code. With TDD, both are written together. TDD is related to the test-first programming.

BDD (behavior-driven development) is a software development process based on the Agile methodology. The concept of behavior-driven development originated from the test-driven development (TDD) approach. This software development process is focused on end-user requirements and their interactions with the product. The concept behind BDD is to enable developers, testers, and business stakeholders to clearly understand an application's behavior and ensure collaboration between technical and non-technical teams.

BDD and TDD may seem very similar since they are both testing strategies for a software application. In both cases, the developer writes the test before writing the code to make the test pass. And in both cases, the tests can be used as part of an automated testing framework to prevent bugs.

- BDD is designed to test an application's behavior from the end user's standpoint, whereas TDD is focused on testing smaller pieces of functionality in isolation. In the prior example, the TDD test asserts the result of a specific method, while the BDD test is only concerned about the result of the higher level scenario.
  <u>*Example*</u>: you can test button click event with BDD
- BDD extends TDD by writing test cases in a natural language that non-technical stakeholders can understand. It encourages collaboration between developers, QA, and non-technical or business participants.

Behavior-driven development represents an evolution beyond TDD, where business goals can be better communicated to developers. By bridging the gap between business and technical teams, BDD helps reduce any confusion about acceptance criteria, identify potential problems with user stories early, and ensure that the application functions as-expected for end users.



```
Feature: User login

  Scenario: Successful login
    Given a user is on the login page
    When the user enters valid credentials
    And the user clicks on the 'Login' button
    Then the user should be redirected to the dashboard
    And the user should see a welcome message
```