۱۰۰۱ نکته مهندسی نرم افزار

نقشه راه معماری، طراحی، توسعه، تست، استقرار، تحویل و نگهداری

ویرایش ۳

گردآورنده : خشایار جام سحر [1]
jamsahar@gmail.com

۱۴۰۴/۰۹/۰۵

2025, Nov 26

این راهنما شامل کلّیات و سرفصلهای مهندسی نرم افزار برای هر یک از مراحل معماری، طراحی، توسعه، تست، استقرار، تحویل و نگهداری است. (و همچنین استانداردها و مستندات مورد نیاز)

[1] - https://www.linkedin.com/in/khashayar-jamsahar-58289743/
[2] - https://www.gnu.org/licenses/gpl-3.0.html

فهرست عناوین

مقدمه

مهمترین رکن مهندسی نرم افزار[3]، دریافت درکی درست از نیازمندیها، و سپس طراحی راه حلی است که به شکلی موثر به آنها پاسخ دهد. این شامل گردآوری، مستندسازی و اعتبار سنجی نیازمندیها و سپس طراحی معماری نرم افزار و ماجولهای آن به شکلی است که قابل **نگهداری**(توسعه قابل برنامه‌ریزی و تکرارپذیر باشد)، **مقیاس پذیر، موثر، قابل اعتماد** و با **کیفیت** باشد. مهندسی نرم‌افزار باید شامل مجموعه‌ای از حوزه‌ها و فرآیندهای کلیدی باشد تا بتواند چرخه‌ی عمر یک نرم‌افزار را از ایده تا پیاده‌سازی و نگهداری به‌صورت ساخت‌یافته مدیریت کند.

فرآیند مهندسی نرم افزار بایستی در زمان مشخص، با هزینه و بودجه مشخص(مقرون به صرفه) و کیفیت قابل قبول، به روش‌های سیستماتیک، قابل اندازه‌گیری و مهندسی‌شده انجام گیرد.

مهندسی نرم‌افزار، نحوه نوشتن راه حل نیست،
بلکه شناسایی آنچه باید در راه حل باشد است.[4]

فعالیتهای مهندسی نرم افزار

فعالیت های توسعه نرم افزار شامل موارد زیر می باشد:

۱) شناسایی (گردآوری، مستندسازی و اعتبار سنجی) نیازمندیها
۲) برنامه ریزی توسعه نرم افزاری
۳) معماری و طراحی نرم افزار
۴) پیاده سازی، آزمایش و مستندسازی
۵) گسترش/توسعه و نگهداری نرم افزار

مهندسی نرم افزار به طور کلی به سه مرحله تقسیم بندی می شود:

مرحله تعریف نیازمندیها و تعیین معماری و طراحی:

بر چیستی تاکید دارد. چه اطلاعاتی باید پردازش شود، کدام کارایی مطلوب است، چه رفتار های سیستمی قابل انتظار است، چه رابطه هایی را می توان برقرار کرد، چه محدودیت هایی وجود دارد و بطور کلی خواسته های کلیدی سیستم شناسایی می شود که شامل موارد زیر است :

۱) طرح ریزی سازمان پروژه (به پیوست ۶ مراجعه شود) و استفاده از استانداردهای موجود[5]
   a. **ISO/IEC/IEEE 15288:2023** (Systems and software engineering — System life cycle processes)
   b. **ISO/IEC/IEEE 12207:2017** (Systems and software engineering – Software life cycle processes)
   c. **The ISO/IEC 25000 series of standards** (System and Software Quality Requirements and Evaluation)[6]
۲) تحلیل نیازمندیها و خواسته ها[7] Software Requirements/Desired و همچنین امکان سنجی/پذیری Feasibility study

---

[3] - https://en.wikipedia.org/wiki/Software_engineering , https://en.wikipedia.org/wiki/History_of_software , https://en.wikipedia.org/wiki/History_of_software_engineering

[4] - https://en.wikipedia.org/wiki/Software_Engineering_Body_of_Knowledge

[5] - **Beynod the mountains are more mountains.**
https://sebokwiki.org/wiki/Systems_Engineering_Related_Standards_Landscape , https://en.wikipedia.org/wiki/ISO/IEC_15288 , https://en.wikipedia.org/wiki/ISO/IEC_12207 , https://sebokwiki.org/wiki/Software_Engineering_in_the_Systems_Engineering_Life_Cycle , https://sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_(SEBoK) , https://connected-corridors.berkeley.edu/guiding-project-systems-engineering-process

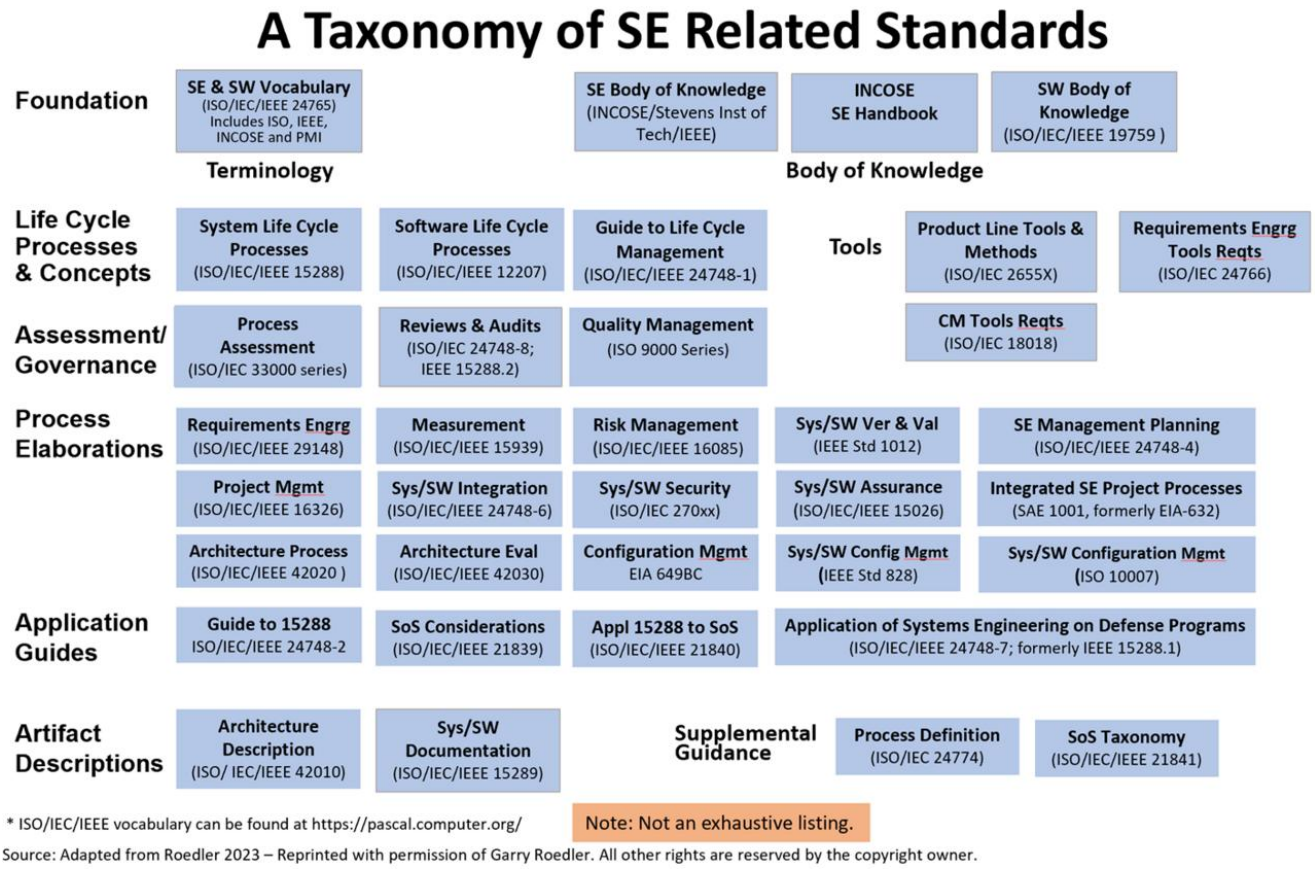
[6] - https://iso25000.com/index.php/en/iso-25000-standards , https://de.wikipedia.org/wiki/ISO/IEC_25000

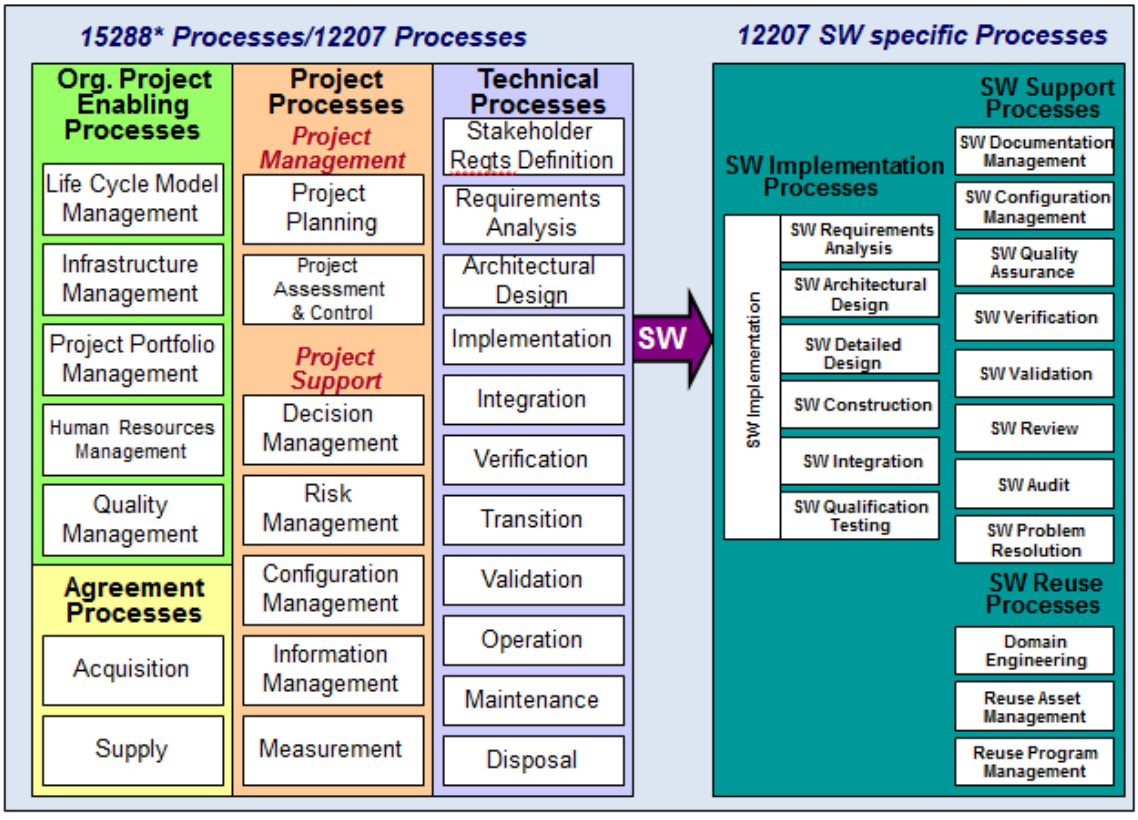
[7] - Steve Jobs:

1) **It is not the customer's job to know what they want.**
2) **Get ahead of the curve.** "It took us three years to build the NeXT computer. If we'd given customers what they said they wanted, we'd have built a computer they would have been happy with a year after we spoke to them – not something they'd want now."
3) **Create and innovate.** "Leaders are fascinated by the future, restless for change, and deeply dissatisfied with the status quo. They are never satisfied with the present, because in their head they can see a better future, and the friction between what is and what could be burns them, propels them forward."
4) **Get closer.** "Get closer than ever to your customers. So close, in fact, that you tell them what they need well before they realize it themselves. "More than Apple listening to us, it's us who listens to Apple."
5) **Build something wonderful.** "Being the richest man in the cemetery doesn't matter to me. Going to bed at night saying 'we've done something wonderful' – that's what matters to me."
6) **Be the expert.** "This is what happens when your MacBook Pro sustains water damage: They are pro machines and they don't like water. It sounds like you're just looking for someone to get mad at other than yourself."

۳) تعیین معماری نرم افزار High Level Design HLD

۴) تعیین طراحی نرم افزار Low Level Design LLD





*Based on 15288:2008. See 15288:2015 for the most recent SE Life Cycle Processes

مرحله توسعه و استقرار Software development :

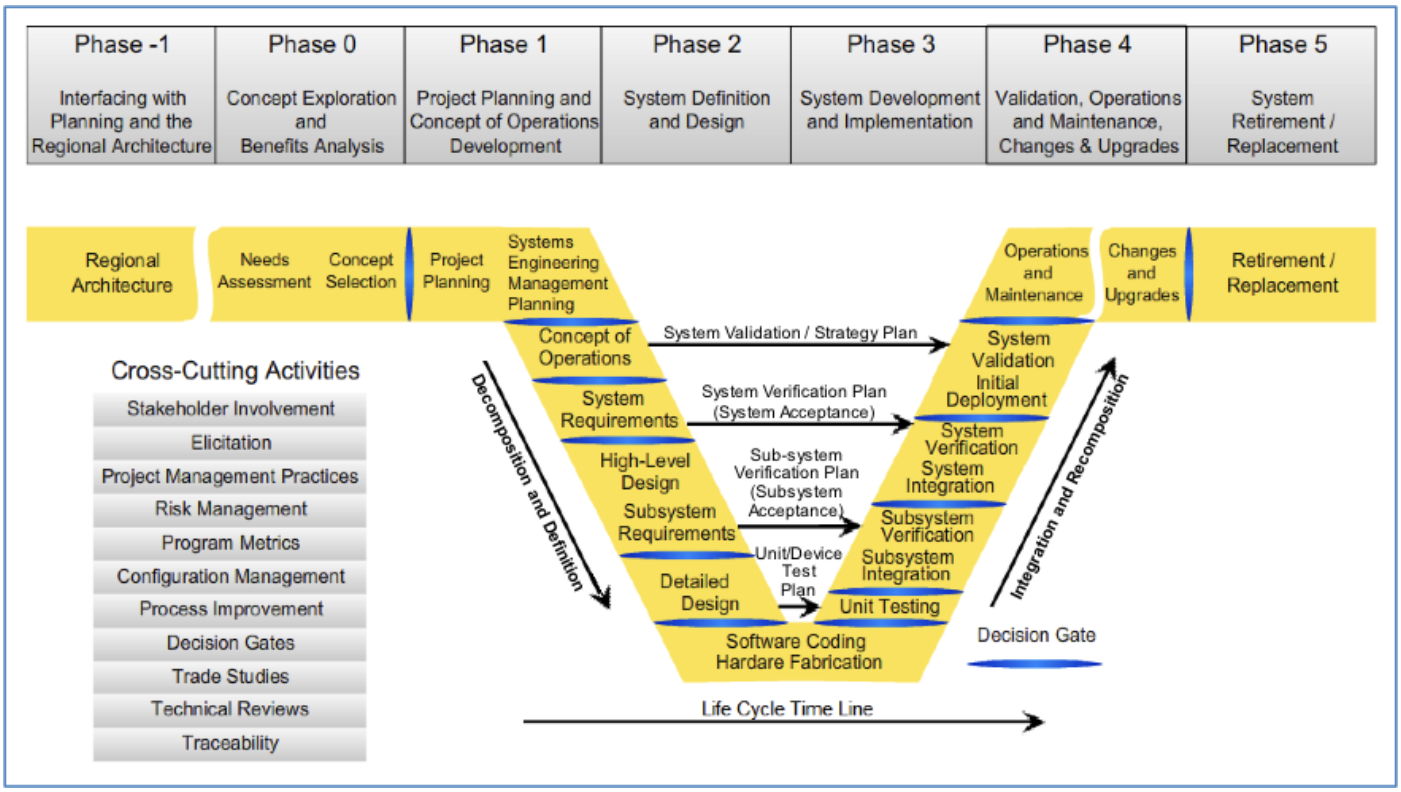
بر چگونگی تاکید دارد. داده ها چه ساختاری داشته باشند، عملیات درون معماری چگونه پیاده سازی می شوند، جزییات روال ها، ویژگی های واسط ها، زبان برنامه نویسی، نحوه آزمایش ها.

سه کار عمده در این فاز شامل :

۱) تولید سورس کد Software development
۲) آزمایش نرم افزار Software testing
۳) استقرار

مرحله نگهداری و پشتیبانی Software maintenance :

بر تغییراتی تاکید دارد که با تصحیحات مورد نیاز در جهت تکامل محیط نرم افزار در ارتباط هستند و همچنین تغییراتی که ناشی از تغییر خواسته های مشتریان/کاربران هستند.

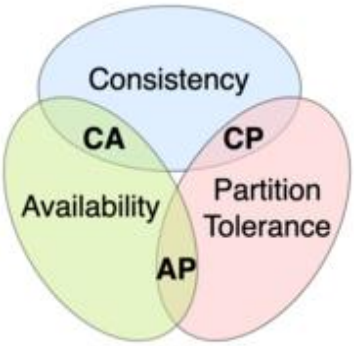


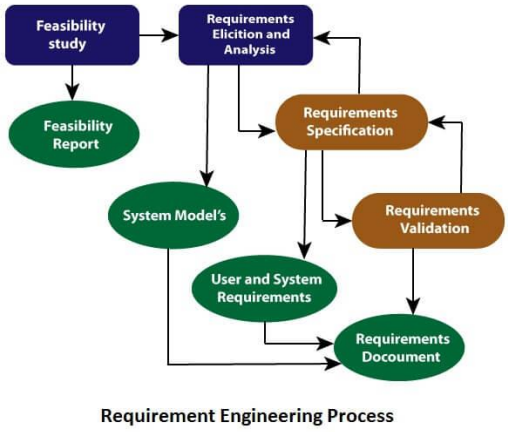
بیزینس کانسپت

۱) بیزینس پلن و سازمان پروژه
   a. ترکیب نیروهای سازمان پروژه
   b. زمان بندی(شکست) پروژه و مایلستون ها
   c. نحوه مدیریت ریسک
۲) اقتصاد نرم افزار، ایده اولیه و نحوه Promoting and Monetizing
۳) انواع نقش ها و کاربران (Admin, Operator, Users, …)
۴) قرارداد(بندها، تبصره ها، پیوستها، فورس ماژور، جرائم و پنالتی ها، …)
۵) سند نیازمندیها[8]: **Software Requirement Specification (SRS)**

[8] - https://en.wikipedia.org/wiki/Requirement , https://en.wikipedia.org/wiki/Requirements_analysis

a. نیازمندیهای کارکردی کاربران/مشتریان[9]
b. نیازمندیهای غیرکارکردی/کیفی[10]
c. نیازمندیهای صاحبان کسب و کار
d. نیازمندیهای معماری و طراحی
e. نیازمندیهای بازار، کسب و کار و همچنین سازمان تنظیم مقررات مربوطه
f. نیازمندیهای سازمان مقرارت Regulatory
g. ...





Requirement Engineering Process

[9] - The function of a system is “what the system is intended to do”.
[10] - The behaviour of such a system is “what the system does to implement its function”.

فاز ۱) تعیین انواع نیازمندیها و Functionality[11]

نیازمندیهای کارکردی (Functional Requirement Specification (FRS): What the system does)

- **کار درست چیست؟** [12]: طراحی فرآیندهای درست برای انجام وظایف (Correctness[13])
- **چگونگی انجام کار؟** [14]: درستی و صحت پروسه ها، وظایف و فعالیتها و نحوه گردش کار
- میزان انسجام/هماهنگی/انطباق (Consistency) طراحی و انطباق پذیری روشها/متدها با فرآیندهای کارکردی

برای پاسخ به موارد فوق بایستی موارد زیر عمیقا بررسی گردد:

۱) درک دامنه بیزینس
۲) تشخیص مشکلات و مسائل موجود
۳) ارزیابی
۴) طبقه بندیها
۵) اولویتها
۶) صحت/اعتبار سنجی
۷) تحلیل سازگاری، رفع ابهام و تناقضات
۸) مدلسازی فرآیندهای کسب و کار BPMN
۹) معماری اطلاعات و داده مورد نیاز
۱۰) Data Flow Diagrams[15] & Interactions
۱۱) Data Dictionaries
۱۲) Entity Relationship Diagrams

> بایستی بخاطر داشت که نسبت موارد Functional به NonFunctional همانند کوه یخ است. نیازمندیهای ***کارکردی*** قسمتی از کوه یخ است که بیرون آب بوده و دیده میشود، در حالیکه نیازمندیهای ***عملکردی/کیفی*** قسمتی است که زیر آب بوده و عموما توجهی به آن نمی شود. مواردی که به عنوان ***بدهی فنی*** در نظر گرفته میشود، عموما در این بخش قرار دارد.

نکته : چنانچه نیازمندیهای کارکردی به درستی پیاده سازی شود (در بهترین حالت)، حاصل یک سیستم **"Functional Prototype"** خواهد بود و چنانچه نیازمندیهای کیفی/عملکردی نیز بدرستی لحاظ گردد، اکنون یک سیستم **"Production-Ready"** **مقیاس پذیر** خواهید داشت.

---

[11] - https://en.wikipedia.org/wiki/COSMIC_functional_size_measurement , https://www.iso.org/standard/38931.html , https://en.wikipedia.org/wiki/The_Simple_Function_Point_method , https://en.wikipedia.org/wiki/IFPUG

[12] - **Validation:** Are we building the right product?

[13] - The software should match all of the customer's needs. Software design should be correct as per requirement.

[14] - **Verification:** Are we building the product, right?

[15] - https://www.scaler.com/topics/software-engineering/data-flow-diagrams/

نیازمندیهای کیفی/عملکردی (Non-Functional Requirement[۱۶]: How the system behaves)

بخش نیازمندیهای کیفی بمراتب مفصل تر از نیازمندیهای کارکردی است و متاسفانه کمتر به آن توجه میشود:

۱) قابلیت نگهداری Maintainability[۱۷]

a) تحلیل پذیری Analyzability
b) سازگار پذیری Adaptability[۱۸]
c) قابلیت تکامل Evolvability[۱۹]
d) چابکی Agility[۲۰]
e) Modifiability[۲۱]

- توسعه پذیری Extensibility[۲۲]
- Portability[۲۳] & Multiplatform
- Changeability[۲۴]

۲) مقیاس پذیری Scalability[۲۵]

a) انعطاف پذیری Flexibility[۲۶]
b) ماجولار Modularity[۲۷]

---

[16] - https://en.wikipedia.org/wiki/Non-functional_requirement , https://learn.microsoft.com/en-us/azure/well-architected/pillars , https://en.wikipedia.org/wiki/Non-functional_requirements_framework , https://en.wikipedia.org/wiki/SNAP_Points , https://en.wikipedia.org/wiki/Architecturally_significant_requirements

[17] - According to the **ISO/IEC 9126 standards** for software quality model, defined according to the standards as "the capability of the software product to be modified". Modifications may include corrections to handle errors, improvements or adaptations in response to changes in the environment and functional requirements. Such modifications could be for operation, maintenance, or evolution purposes.

[18] - Adaptability is the capacity of software to dynamically adjust itself (behaviour, structure or configuration) when reacting to changes in its operating environment in order to keep its services in a good condition. Meanwhile, adaptive maintenance is the "modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment". Adaptation of software system is almost an inevitable process due to the change in customer requirements, the need for faster development of new or maintenance of existing software systems, etc.

[19] - Evolvability is the capacity of software systems to support adaptation and accommodate **long-term changes of new requirements and contexts of use over time with the least possible cost.** The continuously changing stakeholders' requirements make evolvability an important software property to be explicitly addressed throughout the system's lifespan. This property focuses mainly on the long-term evolutionary properties and changes without becoming progressively less useful.

[20] - The ability to respond quickly to changes

[21] - Modifiability is the ability of a system to be easily modified quickly and cost-effective to changes in the environment, requirements or functional specification.

[22] - The ability to acquire new features, deleting unwanted capabilities (to simplify the functionality of an existing application), or restructuring (rationalising system services, modularising, creating reusable components).

[23] - Adapting to new operating environments. Software portability is the capacity to use the same software in different contexts. The key purpose for porting for sales teams is to reach a larger audience. There are different hardware and software platforms available. More users equal more profit.
https://en.wikipedia.org/wiki/Software_portability , https://en.wikipedia.org/wiki/Cross-platform_software

[24] - the capability of the software product to enable a specified modification to be implemented (ISO/IEC 9126 standards). According to this standard, changeability reflects the ability of the software artefact to accept possible future changes, while stability is observed after the change has taken place.

[25] - https://en.wikipedia.org/wiki/Scalability , Allows easy expansion to meet growing demands for compute, storage, and network resource. Able to serve millions of users. It entails the software's capacity to be easily upgraded. Scalability is a crucial feature for any application, and technology enable your code to handle an increasing number of executions on a large scale.

[26] - Flexibility is "the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed". Flexibility is mainly about **future changes of software** and is considered relative to these expected changes, similar to modifiability. Distinguishing it from other properties like adaptivity and changeability, flexibility is defined as "the property of a software system to allow conducting certain changes to the system with **acceptable effort** for modifying the system's implementation artefacts".

[27] - A software product has high modularity if it can be separated into separate independent sections and modified and tested independently.

c) پایائی/پایداری **Stability** (متفاوت است با بخش آزمون پذیری تست پایداری)
- Durability[۲۸]
- Sustainability[۲۹]

d) کشسانی Elasticity and autoscaling[۳۰]

۳) کارا Efficiency[۳۱]

۴) Completeness[۳۲]

۵) درستی Correctness[۳۳]

۶) Dependability[۳۴]

۷) Robustness[۳۵]

۸) Reusability[۳۶]

۹) Interoperability[۳۷]

۱۰) CAP theorem[۳۸]

۱۱) PACELC theorem[۳۹]

۱۲) Encapsulation[۴۰] and Abstraction[۴۱]

---

[28] - **Durability ensures that operations can process highly complex tasks,** run indefinitely, or wait for action for hours or even days without losing data or state. For example, an approval process in a large organization might require multiple levels of sign-offs, and the associated workflow must be able to wait for the necessary approvals, even if it takes several days. With durable workflow engines, developers can create long-running processes that maintain their state and data integrity across extended periods, ensuring that no critical information is lost during the execution.

[29] - Sustainability is defined as “the capacity to endure and preserve the function of a system over an extended period of time”. A sustainable software is “a long-living software system which can be cost-efficiently maintained and evolved over its entire life cycle” and architectural sustainability is “the set of factors that promote an architecture’s stability and longevity during system evolution”.

[30] - Elasticity refers to a system's ability to dynamically adjust its resource allocation (compute, storage, etc.) to match fluctuating demands, scaling up or down as needed. Autoscaling is the process of automatically adjusting resources based on predefined metrics, effectively implementing elasticity. https://en.wikipedia.org/wiki/Autoscaling

[31] - Short response latency. The software must use storage space and time wisely. The software should not make wasteful use of computing devices such as memory, processor cycles, etc.

[32] - The design should have all components like data structures, modules, and external interfaces, etc.

[33] - A software product is correct if the different requirements specified in the SRS Document have been correctly implemented.

[34] - Definitions:
1. Dependability has been considered as “the ability of a system to provide dependable services in terms of availability, responsiveness and reliability”.
2. A widely adopted definition is “the ability to deliver services that can justifiably be trusted in spite of continuous changes”.
3. An alternate definition is “the ability to avoid service failures that are more frequent and more severe than acceptable”.
4. In the context of stability, one can characterise dependability as a kind of behavioural stability that ensures the quality of service provided during operation.

[35] - A system is considered robust if it “retains its ability to deliver service in conditions which are beyond its normal domain of operation”. From a control-theoretic perspective, robustness has been considered as “the property that a system only exhibits small deviations from the nominal behaviour upon the occurrence of small disturbances”.

[36] - Technology should enable developers to create reusable activities and components that can be easily integrated into new Tasks. This not only saves time but also ensures consistent behavior across different processes. Software reusability is a property that refers to a software component's predicted reuse potential. Software reuse not only boosts productivity but also enhances the quality and maintainability of software products.

[37] - The capacity of different solutions to freely and readily communicate with one another is referred to as software interoperability. Capability of 2 or more functional units to process data cooperatively.

[38] - https://en.wikipedia.org/wiki/CAP_theorem

[39] - https://en.wikipedia.org/wiki/PACELC_theorem

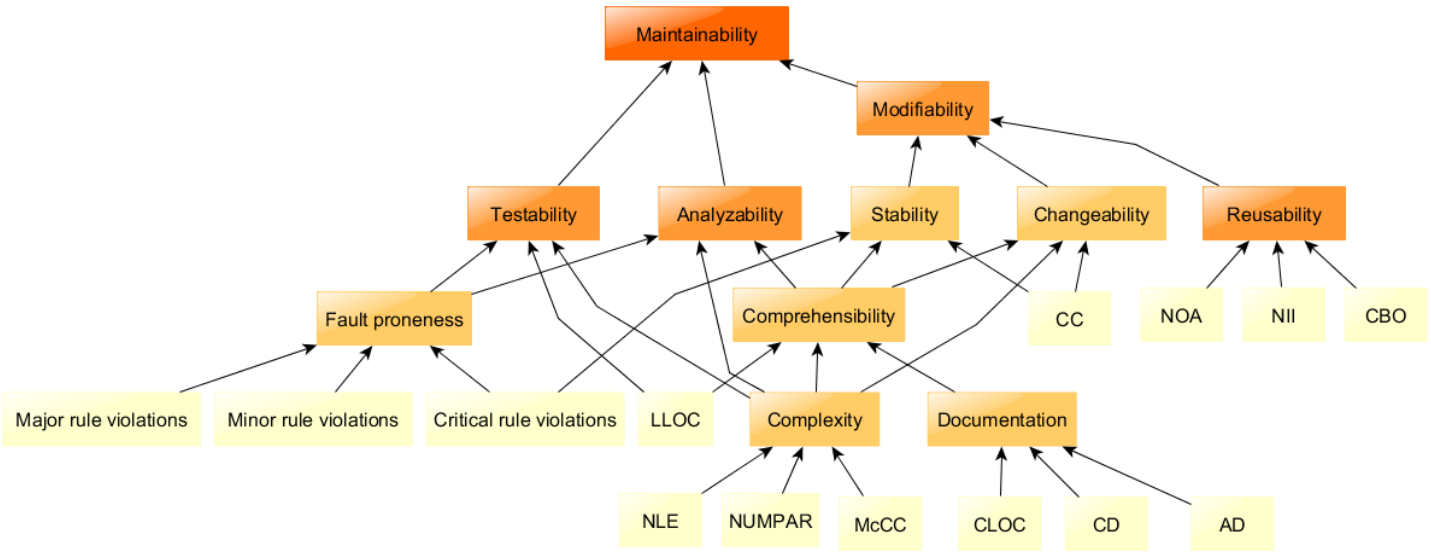
[40] - Each component of the architecture should have a well-defined interface and should not depend on the implementation details of other components. This makes it easier to change or replace individual components without affecting the rest of the system.

[41] -The architecture should use high-level abstractions to represent the different parts of the system. This makes it easier to understand the system and to reason about how it works.

۱۳) مقاومت و تاب آوری Resiliency[۴۲]

۱۴) Fault-Tolerant (تکنیکهای تحمل خطا و ادامه بدون وقفه)

- Fault Detection
- Fault Isolation
- Self Healing
- Chaos engineering[۴۳]
- Health checks
- Idempotent API
- System Monitoring & Alerting
- Observability and Tracing
- Geographic Distribution
- Load Balancing
- Replication
- Redundancy
- Graceful Degradation
- Graceful shutdown
- Load Balancing
- Failover
- Automate Disaster Recovery
- Good Data Backup
- Use Rate Limiting
- Implement Circuit Isolation
- Features
  - Retry Mechanism
  - Time Out
  - Circuit Brekers (Jitter Pattern)
  - Fallback (Default responses)
  - Fault Recovery
  - Bulkheads



۱۵) مدل قابلیت اعتماد/اعتبار Reliability[۴۴] Models

[42] - Enable quick recovery from disruptions, ensuring continuous operations. Ability to recover quickly from a failure/problem/issue/disaster. Applications can automatically recover from failures and handle increased traffic with the help of load balancers and auto-scaling. This feature helps ensure that transient errors, such as temporary network outages, do not cause the entire process to fail.

[43] - https://en.wikipedia.org/wiki/Chaos_engineering , https://principlesofchaos.org/ تست تحمل خطا و پایداری سیستم با شبیه‌سازی شرایط شکست.

[44] - https://www.javatpoint.com/software-engineering-software-reliability-models

1) Reliability has earlier been concerned with “how well the software meets the requirements of the customer”.
2) Following the ISO/IEC/IEEE standards and vocabulary, reliability is “the capability of the software product to maintain a specified level of performance when used under specified conditions” for a specified time.

a) Basic Execution Time Model
b) Jelinski and Moranda Model
- Sukert Modified Schick-Wolverton Model
- Lipow Modified Version of Jelinski-Moranda Geometric Model
- Schick Wolverton Model
- GO-Imperfect Debugging Model
- Jelinski-Moranda Geometric Model
- Little-Verrall Bayesian Model
- Shanthikumar General Markov Model
- Error Detection Model for Application during Software Development
- The Langberg Singpurwalla Model
- Jewell Bayesian Software Reliability Model
- Quantum Modification to the JM Model
- Optimal Software Released Based on Markovian Software Reliability Model
- A Modification to the Jelinski-Moranda Software Reliability Growth Model Based on Cloud Model Theory
- Modified JM Model with imperfect Debugging Phenomenon

c) Goel-Okumoto (GO) Model
d) Musa-Okumoto Logarithmic Model

۱۶) Observability[۴۵]

۱۷) آزمون پذیری[۴۶] Testability

a) Black/white box
b) امنیت[۴۹] & Safety[۴۸] ,Assurance[۴۷]
c) **پایداری[۵۰] Stability**
d) مهندسی قابلیت استفاده[۵۱] Usability و ارزیابی آن
- Understandability
- Learnability
- Operability
- UI/UX

e) High availability[۵۲]

---

3) It is defined as “the continuity of a correct service”, that is the extent to which the system is available when required and behave as expected. The former is measured by corrective action effectiveness, while the latter is measured by the **MTBF (mean time between failure) metric.**
4) Available 24/7 and Continuity of correct service. Means that the software should not fail during execution and be free of flaws. Software reliability is defined as a system's or component's ability to perform its required functions under static conditions for a set amount of time.

[45] - Monitoring and maintaining the health of a system is an essential to keep processes running smoothly. Operations should offer the capability to query and observe the state of every activity, or the entire system. For example, a system administrator can easily track the progress of a long-running data processing tasks, identifying bottlenecks or errors in real-time.

[46] - The software should be simple to test.
https://en.wikipedia.org/wiki/International_Software_Testing_Qualifications_Board https://www.istqb.org/

[47] - https://en.wikipedia.org/wiki/Software_assurance

[48] - https://en.wikipedia.org/wiki/Safety-critical_system , https://standards.nasa.gov/standard/NASA/NASA-STD-87398

[49] - Protect data and systems from cyber threats, avoiding unauthorized access and breaches. The software should protect the data from external threats.

[50] - In software testing, stability testing is an attempt to determine if an application will crash. Stability testing is a method to check the quality and how the system or software behaves in different environmental parameters like temperature, voltage etc. See Stability in Software Engineering: **Survey of the State-of-the-Art and Research Directions** (Maria Salama, Rami Bahsoon, Patricia Lago, 2019 June).

Definitions of Stability:
1) ability to resist to ripple effect of changes
2) ability to remain largely unchanged over time
3) ability to adapt to changes while remaining intact
4) the ability to return to equilibrium state when perturbed from that state
5) the ability to maintain a stated property value or fixed level of operation within specified limits under varying external conditions

[51] - The program should be simple enough for anyone to use.

[52] - https://en.wikipedia.org/wiki/High_availability , https://en.wikipedia.org/wiki/High-availability_application_architecture , https://en.wikipedia.org/wiki/High_availability_software , https://en.wikipedia.org/wiki/High-availability_cluster

f) یکپارچگی Consistency

g) میزان انطباق فرآیندهای کارکردی با نیازمندیهای کاربران نهائی

h) اندازه گیری متریکها[۵۳] (Proces, Project, Product) و میزان تاثیر عناصر بر یکدیگر

- استاتیک/داخلی

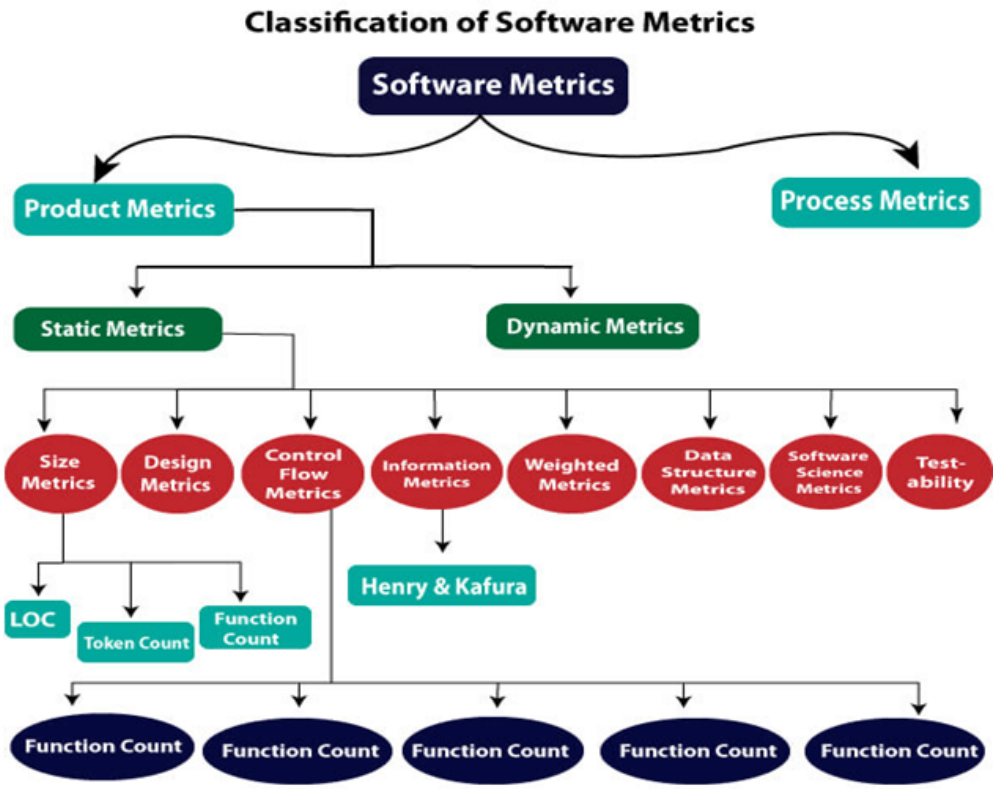
a. LOC (Line of code)/Person/Time[۵۴]
   - KLOC- Thousand lines of code
   - NLOC- Non-comment lines of code
   - KDSI- Thousands of delivered source instruction
b. Number of entities in ER diagram
c. Total number of processes in detailed data flow diagram
d. Function Point Analysis/Count
   - Count the number of functions of each proposed type.
   - Compute the Unadjusted Function Points (UFP).
   - Find the Total Degree of Influence (TDI).
   - Compute Value Adjustment Factor (VAF).
   - Find the Function Point Count (FPC).
e. Halstead's Software Metrics
   - Program length
   - Vocabulary size
   - Program volume
   - Program level
   - Program difficulty
   - Program effort
   - Time to implement
   - Language Level
   - Intelligence Content
   - Programming Time
f. Operational reliability[۵۵]
   - Mean Time to Failure (MTTF)
   - Mean Time to Repair (MTTR)
   - Mean Time Between Failure (MTBF)
   - Mean Up Time (MUT)
   - Mean Down Time (MDT)
   - Rate of occurrence of failure (ROCOF)
   - Probability of Failure on Demand (POFOD)
   - Availability (AVAIL)
   - Maturity
   - Recoverability
   - Software Fault Tolerance
     - Recovery Block
     - N-Version Software
     - N-Version Software and Recovery Blocks

53 - Professors Robert Kaplan and David Norton once said that "if you can't measure it, you can't manage it."
https://en.wikipedia.org/wiki/Software_metric
https://www.scaler.com/topics/software-engineering/software-metrics-in-software-engineering/

54 - Numbers ranging from ~2K-8K LOC/person/year

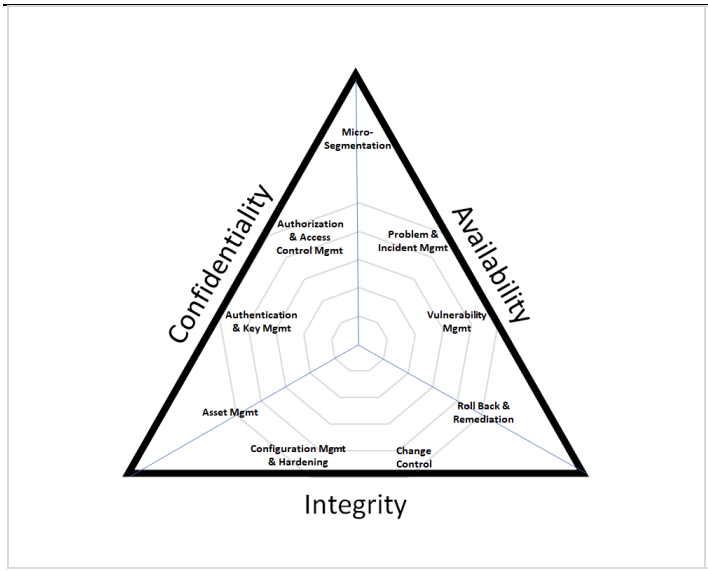
55 - Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and the input are free of error.

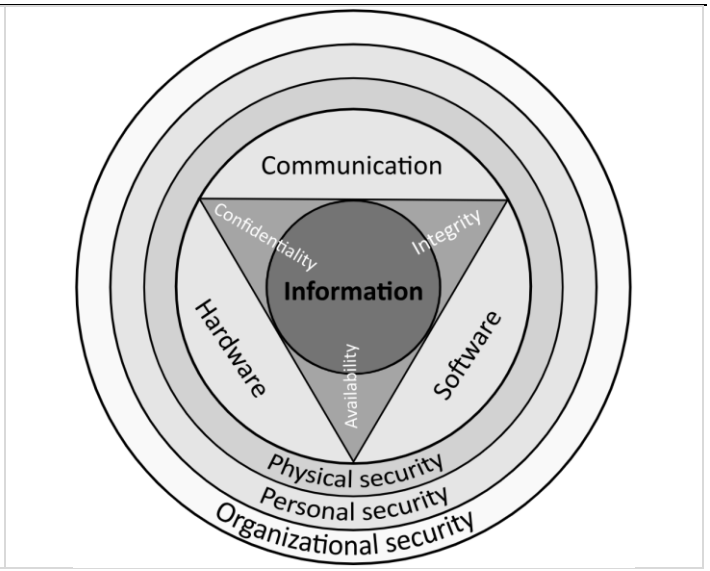
- دینامیک/هیبرید
  a. Cyclomatic Complexity
  b. Extent of class usage
  c. Dynamic Coupling[۵۶]
  d. Dynamic Lack of Cohesion[۵۷]
  e. اندازه گیری پیچیدگی (Complexity[۵۸]) و میزان تاثیر گذاری آن بر بهره وری(Productivity) محصول
  f. پیچیدگی (Complexity) و میزان تاثیر گذاری آن بر قابلیت اعتماد/اعتبار (Rreliability) محصول و ثبات (Stability) آن



۱۸) Service Level Agrements SLA
۱۹) Traffic Patterns
۲۰) Workload Characteristics
۲۱) Data Consistency Models
۲۲) CIA triad
a) محرمانگی Confidentiality
b) یکپارچگی/انسجام Integrity
c) دسترس پذیری Availability

56 - https://en.wikipedia.org/wiki/Coupling_(computer_programming)
57 - https://en.wikipedia.org/wiki/Cohesion_(computer_science)
58 - Cyclomatic complexity (McCabe) M = E – N + 2P https://en.wikipedia.org/wiki/Cyclomatic_complexity





چک لیست امکانات، نیازمندیها و فرآیندها

۱) سند امکانات[۵۹] و نیازمندیها

a. کارکردی(چه کاری انجام شود(فعل)) Functional
b. غیر کارکردی(چگونه آن کار انجام شود(قید)) NonFunctional
   i. تکنولوژی فنی
      ۱. Responsive web applications
      ۲. Single-page web applications
      ۳. Progressive web applications
      ۴. **Real-time[۶۰] web applications**
      ۵. **Reactive[۶۱] web applications** (Asynchronous, Event-based, Non-blocking Architecture)
      ۶. Mashups and web services
   ii. اطلاعاتی
   iii. عملکردی/کارائی
   iv. محیطی(UI/UX)
   v. امنیتی
   vi. کیفی
   vii. مقیاس پذیری
   viii. درستی/یکپارچگی integrity

۲) سند فرآیندهای کسب و کار

a. ورودیها

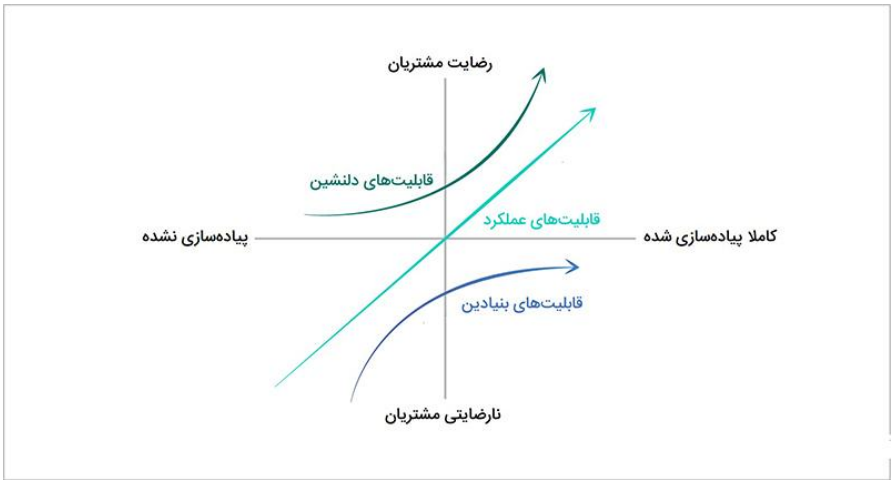
[59] – A feature is a set of logically related requirements that allows the user to satisfy an objective. A feature tends to be a higher-level objective than a requirement.
[60] - A real-time web application delivers responses to events in a measurable and acceptable time period, depending on the nature of the event, by means of asynchronous, bidirectional communication between the client and the server. WebSocket is the core technology employed for developing real-time web applications. WebSocket provides a full-duplex and bidirectional communication protocol over a single TCP connection.
[61] -Reactive Systems are: Responsive, Resilient, Elastic, Message Driven. https://www.reactivemanifesto.org/

b. پروسه ، گردش کار
c. خروجیها

۳) سند گردش کارها Workflows
۴) سند روشها و متدها
a. پاسخهایی که برای چگونگی انجام درست هر کاری لازم است
b. ایجاد و بکارگیری تکنیک های لازم برای اجرای موارد فوق
c. توسعه مجموعه ای از ارتباطات، نیازمندیها، تحلیلها، مدلها، ساختارها، تستها، سرویسها، شاخصها، متریکهای اندازه گیری و ...
۵) ابزارها

می توان از روشهای ذیل برای اولویت‌بندی پروژه ها برای تبیین و تعیین امکانات در روند طراحی محصولات سود جست:

۱) **مدل کانو (Kano Model)**: این روش، ترجیحات مشتری را در ۵ دسته مختلف شامل **ابتدایی، عملکردی، سرگرم‌کننده، بی‌تفاوت و معکوس** دسته‌بندی می‌کند.



۲) **روش RICE**: RICE مخفف Reach, Impact, Confidence, and Effort است. این روش با استفاده از ۴ فاکتور **دسترسی، تاثیر، اطمینان و تلاش** امتیازی را برای هریک از امکانات مد نظر محاسبه می‌کند و در اولیت‌بندی آنها به شما کمک خواهد کرد. این روش از تیم‌های توسعه محصول می‌خواهد که پیش از کمیت‌سنجی معیارهای محصول خود، از رویکرد **SMART** استفاده کنند. بنابر روش SMART، رویکرد شما باید **واضح** (Specific)، **قابل اندازه‌گیری** (Measurable)، **قابل دست‌یابی** (Attainable)، **مرتبط** (Relevant) و همانطور که در پارامتر امتیاز **"دامنه"** مشاهده میشود، **متکی بر زمان** (Time-Based) باشد.

a. دامنه Reach: قابلیت مورد نظر در یک بازه زمانی مشخص قرار است چه تعداد از کاربران را تحت‌الشعاع خود قرار دهد؟
b. اثر گذاری Impact: این قابلیت چقدر روی تجربه هر کاربر تاثیر می‌گذارد؟
c. اطمینان Confidence: چقدر از امتیاز دامنه و اثرگذاری خود اطمینان داریم؟ چقدر داده در اختیار داریم که تخمین‌هایمان را تایید کنند؟
d. تلاش Effort : میزان انرژی و تمرکز لازم از سوی کارکنان روی یک پروژه را تعریف می‌نماید. از منظر توسعه محصول و طراحی، این نوآوری یا ویژگی نیازمند چه میزان سرمایه‌گذاری زمانی است؟

$$\text{امتیاز RICE} = \frac{\text{دامنه} \times \text{اثرگذاری} \times \text{اطمینان}}{\text{تلاش}}$$

۳) **ماتریس تلاش-تاثیر:** این روش که با عنوان ماتریس ارزش-پیچیدگی نیز شناخته می‌شود، ابزاری نسبتا ساده برای اولویت‌بندی است. بر مبنای این ماتریس دو وجهی، امکانات مختلف بر مبنای میزان تلاش مورد نیاز برای توسعه یک عملکرد و تاثیر آن در چهار دسته مختلف طبقه‌بندی می‌شوند.

۴) **روش MoSCoW:** این روش که بیشتر در توسعه نرم‌افزار کاربرد دارد، نیازها را در ۴ دسته **حتما باید باشد، باید باشد، می‌تواند باشد و نیازی نیست باشد** طبقه‌بندی می‌کند. این روش، روشی مناسب برای اولویت‌بندی نیازهای حساس به زمان در چارچوب زمانی ثابت است تا اطمینان حاصل شود که مهم‌ترین قسمت‌ها زودتر توسعه داده می‌شوند.

۵) **اولویت‌بندی WSJF:** این روش که نام آن مخفف عبارت «Weighted Shortest Job First» است، معادله‌ای متشکل از ۴ مولفه برای رتبه‌بندی امکانات است. این مولفه‌ها شامل **ارزش کاربر-بیزینس، بحران زمانی، "کاهش ریسک و یا ایجاد شانس" و حجم کار** است. با محاسبه این فاکتورها برای نیازهای مختلف، می‌توان آنها را اولویت‌بندی کرد.

۶) **ارزش در برابر تلاش :** این رویه شامل انبوهی حدس و گمان و ارائه نظرات گوناگون می‌شود تا بتوان در نهایت به بزرگ‌ترین سوال در فرایند اولویت‌بندی پاسخ داد: این‌که فلان قابلیت یا به‌روزرسانی، در نهایت به بهبود وضعیت اهداف و معیارهای محصول منجر می‌شود یا خیر؟ و این‌که آیا می‌توان با منابعی که همین حالا در اختیار داریم، همان قابلیت یا به‌روزرسانی را بیش از پیش گسترش دهیم؟

چقدر اهمیت دارد؟	توسعه آن چقدر دشوار است؟

• ارزش و درآمد بالقوه
• مزایا
  ○ برای مشتریان کنونی
  ○ برای مشتریان بالقوه
• اثر گذاری
  ○ بر اهداف کسب و کار
  ○ برا اهداف استراتژیک
	• هزینه
	• تلاش
	• ریسک
	• پیچیدگی

۷) **نقشه داستانی :** زیبایی این بستر اولویت‌بندی توسعه محصول، سادگی آن است. نقشه داستانی Story Mapping ضمنا تمام تمرکز را بر تجربه کاربران می‌گذارد. این رویه با محوریت تجربه کاربر دنبال می‌شود و در واقع می‌توانید برای مشتریان داستان‌سرایی کنید. نقشه داستانی باعث می‌شود به سرعت و با بیشترین بهره‌وری قادر به شناسایی «کمینه محصول پذیرفتنی» (یا MVP) خود باشید. متاسفانه در این رویکرد توجهی به فاکتورهای اولویت‌بندی توسعه محصول مانند "ارزش کسب‌وکار و پیچیدگی‌های فرایند توسعه" نمی‌شود.

۸) **درخت محصول :** این بستر اولویت‌بندی توسعه محصول در واقع یک بازی است که توسط شخصی به نام بروس هولمن طراحی شد. این فعالیت بر شکل دادن بر محصول تمرکز دارد، به‌گونه‌ای که با خروجی‌های مد نظر مشتریان سازگار باشد و بیشترین ارزش‌سازی را برای ذینفعان به همراه آورد. هدف این بازی، هرس کردن درخت محصول و موارد باقی‌مانده در لیست وظایف است تا بتوان اطمینان حاصل کرد هیچ ایده‌ای نادیده گرفته نمی‌شود.

۹) **ماتریس آیزنهاور:** این ماتریس نیز به شکلی است که با طبقه‌بندی امکانات و نیازها در ۴ دسته مختلف شامل **انجام دادن، زمان‌بندی، حواله کردن و حذف کردن** مهم‌ترین امکانات برای طراحی و توسعه را مشخص می‌کند. به عبارت دیگر از مدیریت زمان ۴ ربعی زیر سود میبرد:

a. فوری و مهم
b. فوری اما نه مهم
c. مهم اما نه فوری
d. نه فوری و نه مهم

فاز ۲) تعیین معماری و نحوه طراحی و مهندسی نرم افزار

به شکل کلی، طراحی و مدل سازی[۶۲] معمولا دارای سه سطح به شرح ذیل می باشد :

۱) **سند طراحی/مدلسازی مفهومی**[۶۳]: یعنی معماری **ایده آل**، انتزاعی و ذهنی از هدف مورد نظر است که صرفا اجزای اکو سیستم، علل، مرزبندی/دامنه، وجود ارتباط بین اجزا و چارچوبها را نمایش میدهد. اجزای اکو سیستم در این مرحله دارای **هیچ وزن و یا مشخصه کمّی/کیفی، کارکردی/عملکردی، تکنولوژیک** نمی باشند و صرفاً حضور یا عدم حضور آنها اهمیت دارد.

۲) **سند طراحی/مدلسازی منطقی** : این بخش شامل **کلیه حقایق (و ایده آل های)** اکوسیستمی است که بایستی به آنها پرداخته شود. در این مدل، اجزای اکو سیستم، بهمراه مشخصه های **کمّی، کارکردی(Functional)**، نحوه ارتباط بین اجزا(یک طرفه، دوطرفه، کدینگ) و وظایف و نتایج مورد انتظار طراحی می شود. این مدل نسبت به طراحی مفهومی، دارای سطح پائین تری بوده و ارتباط اجزا و سیستم ها و همچنین از نظر تکنولوژیک کاملا بایستی منطقی، عقلانی و در دسترس بوده و از نظر موفقیت های تجاری قابل پذیرش باشد.

۳) **سند طراحی/مدلسازی فیزیکی(Detail Design)** : در این مدل، اجزای اکو سیستم، بهمراه مشخصه های **کیفی، عملکردی(Non Functional)**، و استانداردهای مربوطه طراحی می شود. طراحی بایستی به شکلی باشد که شامل مشخصات کلیه ورودی ها، خروجی ها، فرآیندها، زمانبندی و همچنین شیوه ارتباط با سایر اکوسیستم های مجاور و مرتبط در مجموعه باشد. خروجی طراحی این بخش بایستی کاملا منطبق با آنچه که قرار است **واقعیت** یابد و عملیاتی شود، **با جزئیات کامل** تهیه و ارائه گردد.

بدیهی است که در طول دوره رشد و بلوغ اکوسیستم هریک از این بخش ها نیز به نوبه خود غنی تر خواهد شد. طراحی و مدلسازی می تواند بالا به پائین و یا پائین به بالا[۶۴] بوده، و البته معمولا ترکیبی از این دو می باشد.

طراحی نرم افزار شامل مستندات زیر می باشد:

۱) **سند معماری(اطلاعات) سازمانی**[۶۵] : پیشتر توسط سازمان بایستی تهیه شده باشد و کلیه نرم افزارها بر اساس آن طراحی و توسعه داده می شود. این سند شامل معماری و طراحی فرآیندها، ساختار داده ها و جریان اطلاعات بین واحدها می باشد.

۲) سند حاکمیت/حکمرانی[۶۶] داده[۶۷] Data Governance

۳) سند استانداردهای پایه ای تولید و توسعه نرم افزار[۶۸]:

a. استاندارد سند توصیف متدولوژی MDD
b. استاندارد طرح مدیریت پروژه PMP

---

62 - https://en.wikipedia.org/wiki/Modeling_Maturity_Levels , https://en.wikipedia.org/wiki/Model-based_systems_engineering https://en.wikipedia.org/wiki/Story-driven_modeling , https://en.wikipedia.org/wiki/Model-driven_engineering , https://en.wikipedia.org/wiki/Domain-specific_modeling , https://en.wikipedia.org/wiki/Domain-specific_language , https://en.wikipedia.org/wiki/Domain-specific_multimodeling , https://en.wikipedia.org/wiki/Model-driven_architecture

۶۳ – البته در برخی موارد بقدری پیشرفته، آرمانی و ایده آل است که عملا با تکنولوژی فعلی و امکانات حال حاضر به سختی قابل اجرا است یا مقرون به صرفه و تجاری نیست.

64 - https://en.wikipedia.org/wiki/Bottom-up_and_top-down_design , https://en.wikipedia.org/wiki/Agent-based_model

۶۵ – IT Enterprise Architecture.

۶۶ – حاکمیت و حکمرانی به این معناست که چه کسی قدرت و اختیار دارد، در چه حدی و درباره چه چیزی، چه تصمیماتی میتوان گرفت، چه مسئولیتی می پذیرد، چگونه بایستی پاسخگو باشد، چگونه عواقب تصمیمات را بر عهده بگیرد، ... . (بایستی به خاطر داشت که یکی از اهداف نرم افزار تولید داده با کیفیت می باشد.)

67 - https://en.wikipedia.org/wiki/Data_governance

۶۸ – مستندات کاملتر در سایت **شرکت مهندسی نرم افزار گلستان** http://golsoft.com/pages/fa/fa-namatan.html **نماتن** .

ایضاً http://old.irannsr.org/web_directory/ کتابخانه اسناد **سازمان نظام صنفی رایانه ای کشور**

https://www.irannsr.org/fa/page/120024-نماتن.html

https://www.irannsr.org/fa/page/120032-اسناد-نماتن-نسخه-قدیمی.html

https://www.irannsr.org/fa/page/120208-1402-اسناد-نماتن-بازنگری-شده.html

c. استاندارد طرح تضمین کیفیت QAP
d. استاندارد طرح مدیریت پیکربندی CMP
e. استاندارد طرح تصدیق و صحه گذاری V&V
f. استاندارد طرح آزمون نرم افزار
g. استاندارد طرح انتقال و تحویل نرم افزار
h. استاندارد طرح ضمانت نرم افزار
i. استاندارد طرح نظارت

۴) سند معماری[۶۹] سیستم یا **High-Level Design** : (برای اطلاعات بیشتر به پیوست ۱ مراجعه شود)

انتخاب معماری درست برای سیستم شما می تواند تفاوت بین "موفقیت بلند مدت" و "کابوس ریفکتورینگهای بی پایان" باشد. معماری، ساختار کلی طراحی سیستم را نشان میدهد. در طراحی سطح بالا، تمرکز صرفا بر چگونگی تعامل اجزای مختلف سیستم با یکدیگر می باشد (بدون وارد شدن به جزئیات داخلی و کدینگ اجزا). این بخش کمک میکند تا افرادی که درگیر پروژه هستند به درک بهتری از اهداف هر جز دست یابند.

> معماری خوب آنست که سیستم را تا حد امکان به شکلی مدلسازی و طراحی نماید که لایه **"منطق و قواعد کسب و کار"** مستقل از سایر لایه ها/اجزا/عناصر (تکنولوژی، زیرساخت، پلاتفرم، فریم ورک، دیتابیس، واسط کاربری، سرویسها، APIs، ...) بوده و متاثر از تغییرات آنها نباشد. چنین معماری/طراحی سبب خواهد شد تا هر لایه مستقل از سایر لایه ها، توسعه داده شده و تست شود.

اجزای طراحی سطح بالا، میتواند شامل موارد زیر باشد :

- **System Architecture:** System architecture is an overview of the entire system which represents the structure and the relationships between various components. It helps to visually represent how different parts interact and function.
- **Modules and Components:** High-Level design breaks down the systems into modules or components each with specific roles and responsibilities, and has a distinct function that contributes to entire system helping in developing an efficient system.
- **Data Flow Diagrams (DFDs):** Data Flow Diagrams demonstrates the data movement within the system. They help to understand how information is processed and handled.
- **Interface Design:** This component focuses on how different modules communicate with one another. It details the application programming interfaces **(APIs)** and user interfaces necessary for seamless interaction between components.
- **Technology Stack:** The technology stack are various technologies and tools that will be used in the development of the system. This includes **programming languages, frameworks, databases.**
- **Deployment Architecture:** It includes how the system will be hosted and accessed. It includes server configurations, cloud infrastructure, and network considerations.

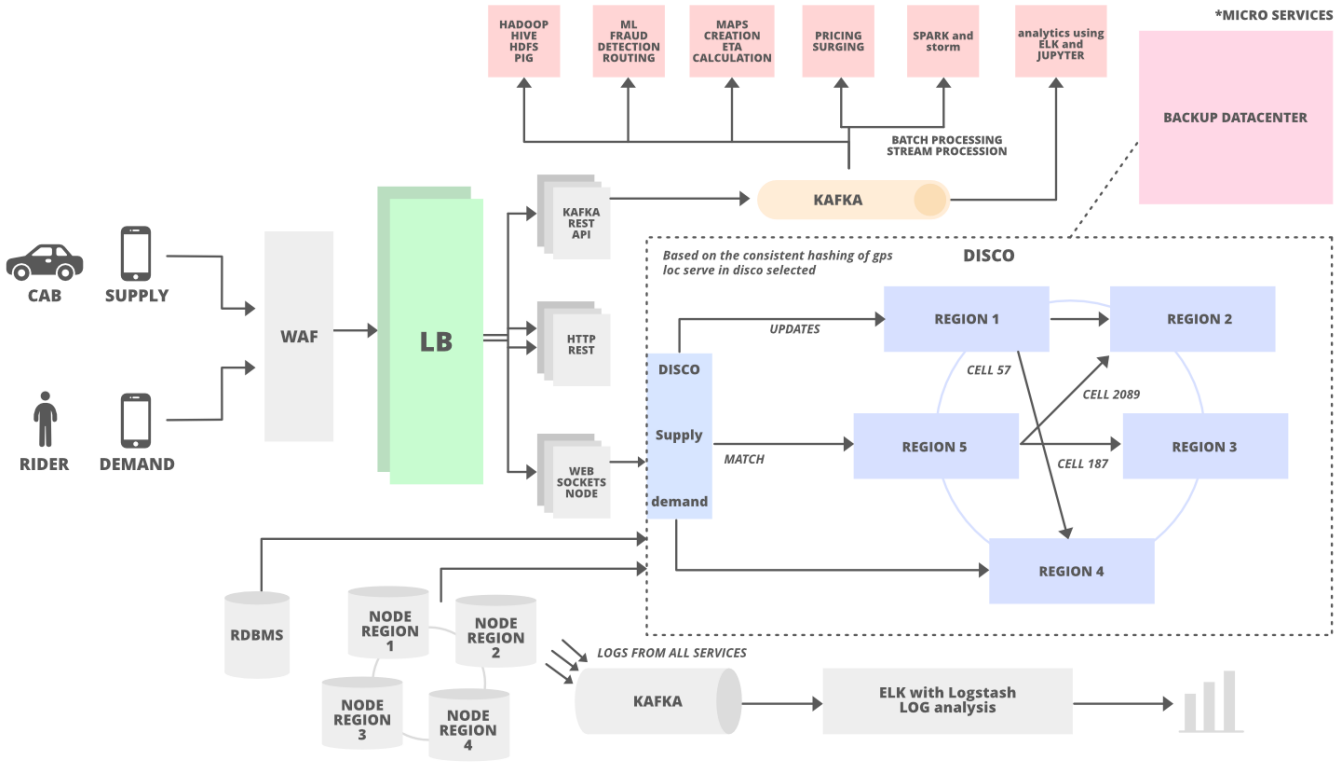
[69] - https://en.wikipedia.org/wiki/Software_architecture , https://en.wikipedia.org/wiki/High_Level_Architecture , https://cio-wiki.org/wiki/Software_Architecture , https://cio-wiki.org/wiki/Architecture , https://pitchtechnologies.com/wp-content/uploads/2020/06/TheHLAtutorial.pdf , https://learning-notes.mistermicheels.com/architecture-design/architectural-boundaries/ https://github.com/mistermicheels/learning-notes/blob/master/architecture-design/Architectural-Boundaries.md https://github.com/donnemartin/system-design-primer

ISO/IEC/IEEE Standard that defines software architecture as the “fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution”.

System's architecture defines shape of the system:

- How the system is divided into components
- How those components are arranged
- What kinds of boundaries exist between different components
- How components communicate across those boundaries

Fundamentals of Software Architecture (Mark Richards & Neal Ford, O’Reilly, 2020),
Software Architecture Patterns 2nd Ed (Mark Richards, O’Reilly, 2022),
Software Architecture: The Hard Parts (Mark Richards, O’Reilly, 2022),
https://www.developertoarchitect.com/lessons/



برای پیاده سازی و اجرای معماری (طراحی سطح بالا) نیز می توان از الگوهای[۷۰] زیر سود جست :

Software Architectural Styles[71]

- **Actor-based Model**[72] (Distributed[73] Shared nothing[74] Concurrent programming[75] via message passing)
- Big Ball of Mud Antipattern (Coupling everythings)
- Broker architecture pattern
- Cell-Based Architecture[76]
- Centralized, Decentralized **(Distributed Event-Based Systems)**
- Client–server (1990s)[77]
- **Clean Architecture** (2011)[78]
- Component-based
- Controller Responder Pattern (Master/Slave)
- Data-centric
- **Domain-Driven Design (DDD)**(2003)[79]

[70] - https://en.wikipedia.org/wiki/List_of_software_architecture_styles_and_patterns
https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff650706(v=pandp.10)

[71] - https://learn.microsoft.com/en-us/azure/architecture/browse/ , https://cio-wiki.org/wiki/Architectural_Style ,
https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/

[72] - https://en.wikipedia.org/wiki/Actor_model ,
https://github.com/denyspoltorak/publications/tree/main/IntroductionToSoftwareArchitectureWithActors

[73] - https://en.wikipedia.org/wiki/Distributed_computing , https://www.youtube.com/@6.824/videos

[74] - https://en.wikipedia.org/wiki/Shared-nothing_architecture

[75] - https://en.wikipedia.org/wiki/Concurrent_computing

[76] - https://github.com/wso2/reference-architecture/blob/master/reference-architecture-cell-based.md#section-1-abstractions

[77] - https://www.geeksforgeeks.org/system-design/client-server-architecture-system-design/

[78] - https://blog.cleancoder.com/uncle-bob/2011/11/22/Clean-Architecture.html

[79] - A domain is "an area of knowledge or activity". https://en.wikipedia.org/wiki/Domain-driven_design ,
https://www.geeksforgeeks.org/system-design/domain-driven-design-ddd/ , https://en.wikipedia.org/wiki/Domain_engineering ,
https://en.wikipedia.org/wiki/Domain_analysis
Domain: Bounded Contexts, Business Function Areas (Orders, Shipping, Invoicing, Accounting, …)

    - **Screaming Architecture**[80] (Robert C. Martin, "Uncle Bob.")
- Event-Bus Pattern
- **Event Driven Architecture (EDA)** [81] (Event Driven Design & Message Driven Design)
    - Application Internal (etc, Goroutins in Golang, Akka)
    - Ephemeral (ESB (Enterprise Service Bus))
    - Queues (JMS) (etc, RabbitMQ, Active MQ)
    - Pub/Sub (Streaming[82]) (etc, Apache Kafka)
- Feature-based Architecture[83] (Vertical Slice Architecture)[84]
- Fractal Architectures[85] (A Software Craftsman's take to Infrastructure as Code **IaC**)
- **Hexagonal Architecture** (also known as Ports and Adapters) (2005)[86]
- Layered architecture, **N Layered (or multilayered and Segmented[87]) architecture**[88]
  Three-tier, Multi-Tier Pattern (Application, API, Product Services, Common Services, Data)[89]
- Microkernel (Core) + Plug-ins architecture
- **Microservices architecture**
- Modular[90] Monolith[91]
- Orchestration-Driven Service-Oriented Architecture (SOA)
- **Onion Architecture** (2009)
- Pipes and Filters architecture[92]
- Primary-Replica Architecture Pattern
- Resource-oriented architecture[93]
- Rule-based
- Serverless Architectures (BaaS (Backend as a Service) and FaaS (Function as a Service))
- **Space-based architecture**[94] (SBA)(Shared Nothing architecture & distributed computing architecture)
- **Shared Nothing architecture (SNA)**

Software Architectural Patterns[95]

- Batch request (Request Bundle pattern)
- Blackboard (design pattern)

[80] - https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html

[81] - به پیوست ۱ مراجعه شود

[82] - A stream is the set of messages that form an HTTP request/response pair. The term stream comes from HTTP/2.
https://github.com/AutoMQ/automq/wiki/Differences-Between-Messaging-Queues-and-Streaming

[83] - https://en.wikipedia.org/wiki/Feature-driven_development , https://en.wikipedia.org/wiki/Feature_model

[84] - https://en.wikipedia.org/wiki/Vertical_slice , https://github.com/Amitpnk/Vertical-Slice-Architecture ,
https://www.milanjovanovic.tech/blog/vertical-slice-architecture , https://treblle.com/blog/minimal-api-with-vertical-slice-architecture ,
https://nadirbad.dev/vertical-slice-architecture-dotnet

[85] - https://medium.com/yanchware/fractal-architectures-a-software-craftsmans-take-to-infrastructure-as-code-8a7f81192110

[86] - https://en.wikipedia.org/wiki/Hexagonal_architecture_(software)

[87] - https://github.com/wso2/reference-architecture/blob/master/reference-architecture-layered-segmented.md

[88] - **Layered architecture,** also known as **n-tier architecture** is one of the most commonly used patterns for software development.

[89] - https://en.wikipedia.org/wiki/Multitier_architecture

[90] - In summary, a module can be defined as a "logical container that groups related functionality together and makes functionality accessible via well-defined contracts.". Below are the characteristics of a module:

- Must be independent and interchangeable.
- Must contain everything necessary to serve the functionality independently.
- Must expose well-defined contracts/interfaces for other modules to consume the functionality.

[91] - Modular monolithic architecture is a way of organizing a software application into a set of independent modules with well-defined boundaries. These modules have specific functionality, which can be independently developed and tested, while the entire application is deployed as a single unit. https://static.simonbrown.je/modular-monoliths.pdf

[92] - https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters

[93] - https://en.wikipedia.org/wiki/Resource-oriented_architecture

[94] - https://en.wikipedia.org/wiki/Space-based_architecture

[95] - https://cio-wiki.org/wiki/Architectural_Principles , https://cio-wiki.org/wiki/Architectural_Pattern ,

- Circuit Breaker[96]
- Claim-Check pattern
- Competing Consumers pattern
- **CQRS[97] + Event Sourcing[98]**
- Inbox and outbox pattern
- Model-View-Controller (MVC)
- Peer-to-peer (P2P) architecture pattern
- Publish-Subscribe[99] messaging pattern
- Queue-Based Load Leveling
- Request-Response (REST, RPC, SOAP, …)
- Rule-based
- Saga pattern
- Shared-Data pattern
- Strangler fig pattern
- Backpressure and Throttling[100]

Architecture styles rating summary

	Layered	Microkernel	Event-driven	Microservices	Space-based
Partitioning	T	D/T	T	D	T
Overall cost	$	$	$$$	$$$$$	$$$$$
Agility	●	●●●	●●●	●●●●●	●●
Simplicity	●●●●●	●●●●	●	●	●
Scalability	●	●	●●●●●	●●●●●	●●●●●
Fault tolerance	●	●	●●●●●	●●●●●	●●●
Performance	●●●	●●●	●●●●●	●●	●●●●●
Extensibility	●	●●●	●●●●●	●●●●●	●●●

Pattern-analysis summary

	Layered	Event-driven	Microkernel	Microservices	Space-based
Overall Agility	↓	↑	↑	↑	↑
Deployment	↓	↑	↑	↑	↑
Testability	↑	↓	↑	↑	↓
Performance	↓	↑	↑	↓	↑
Scalability	↓	↑	↓	↑	↑
Development	↑	↓	↓	↑	↓

مهمترین بخش معماری یا طراحی سطح بالا این است که

چگونه سیستمی مقیاس پذیر طراحی کنیم؟!

برای پاسخ به سوال فوق، بر روی گلوگاه های سیستم و موارد زیر بایستی تمرکز نمود :

1) Modularity, Separation of Concerns, Decoupling and Loose Coupling
2) Load Distribution, Horizontal Scalability (Scaling Out) (Elastic Infrastructure).
3) Geographic distribution: Deploy the system in multiple geographic regions to be closer to users.
4) Data partitioning, Scalable Data Sharding
5) Concurrency and Parallelism
6) Utilize technologies that support auto-scaling and load balancing
7) Software Architecture Scalability: Microservices Architecture, Event-Driven Architecture, Space-Based Architecture, **CQRS**+Event Sourcing, Distributed Systems, Serverless Arch, …

96 - https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker
97 - https://en.wikipedia.org/wiki/Command_Query_Responsibility_Segregation , https://eventuate.io/
98 - https://martinfowler.com/eaaDev/EventSourcing.html , https://www.dremio.com/wiki/event-sourcing/
https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing#when-to-use-this-pattern
99 - https://en.wikipedia.org/wiki/Publish-subscribe_pattern
https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber
100 - https://learn.microsoft.com/en-us/azure/architecture/patterns/throttling

8) Single Point of Failure (**SPOF**[101]) in System Design
    a. Common Single Points of Failure in Systems
        i. Databases
        ii. Application Servers
        iii. Load Balancers
        iv. Network Connections
    b. Strategies to Eliminate Single Points of Failure
        i. Redundancy
        ii. Availability Zones and Failover Mechanisms
        iii. Load Balancing
        iv. Geographic Distribution
        v. Monitoring and Alerts
            1. Health monitoring
            2. Availability monitoring
            3. Performance monitoring
            4. Secuity monitoring
            5. Usage monitoring
9) Fault Tolerance & Resilience
10) Asynchronous Processing: Use message queues and asynchronous tasks to handle operations that don't require immediate responses
11) Statelessness: Design services to be stateless, meaning they don't store session data between requests. This simplifies scaling and load balancing.
12) Cloud-Native Design[102] & Resource Management
13) Performance Optimization
14) Maintainability & Extensibility
15) Capacity Estimation
16) Caching and Content Delivery Networks (**CDN**s)[103]
17) Prefer simple **CAP** systems over exotic eventual consistency if possible.
18) Observability, Monitoring and Performance Optimization (identify bottlenecks)
19) **Tradeoffs and Priorities**

> **Bottlenecks[104]:** Any shared resource in your system is potentially a bottleneck that will limit capacity. Databases, Disk I/O, message queues, long latency network connections, thread and connection pools and shared microservices are all prime candidates for bottlenecks. The data tier is the hardest to scale.

۵) سند طراحی سطح پائین **Low-Level Design** : (برای جزئیات بیشتر به پیوستهای ۳،۲ و ۴ مراجعه شود)

در این مرحله طراحی سطح بالا، تبدیل به کلاسها، الگوریتمها، ساختمان داده، واسط ها، الگوهای طراحی، بهروشها ، API، Object-level Design، با جزئیات بیشتری خواهد شد. از مزایای این بخش میتوان به موارد زیر اشاره نمود:

- این مرحله کمک میکند تا درک بهتری از چگونگی و نحوه تعامل از هر بخش نرم افزار داشته باشیم.
- طراحی با ساختار بهینه، کمک میکند تا هر بخش سیستم براحتی به روز آوری شود، بدون اینکه سایر بخشها اثر پذیرند.

[101] - Few reasons why SPOF is such a significant concern:
- **Reliability:** A single failure can take the entire system down, which could mean business losses and user dissatisfaction.
- **Scalability:** Systems with SPOFs often struggle to scale, as each component adds risk.
- **Security :** A single vulnerable entry point makes it easier for attackers to compromise the whole system.

[102] - https://github.com/wso2/reference-architecture/blob/master/digital-enterprise-k8s-wso2-api-platform.md
https://github.com/wso2/reference-architecture/blob/master/reference-cloud-native-architecture-digital-enterprise.md

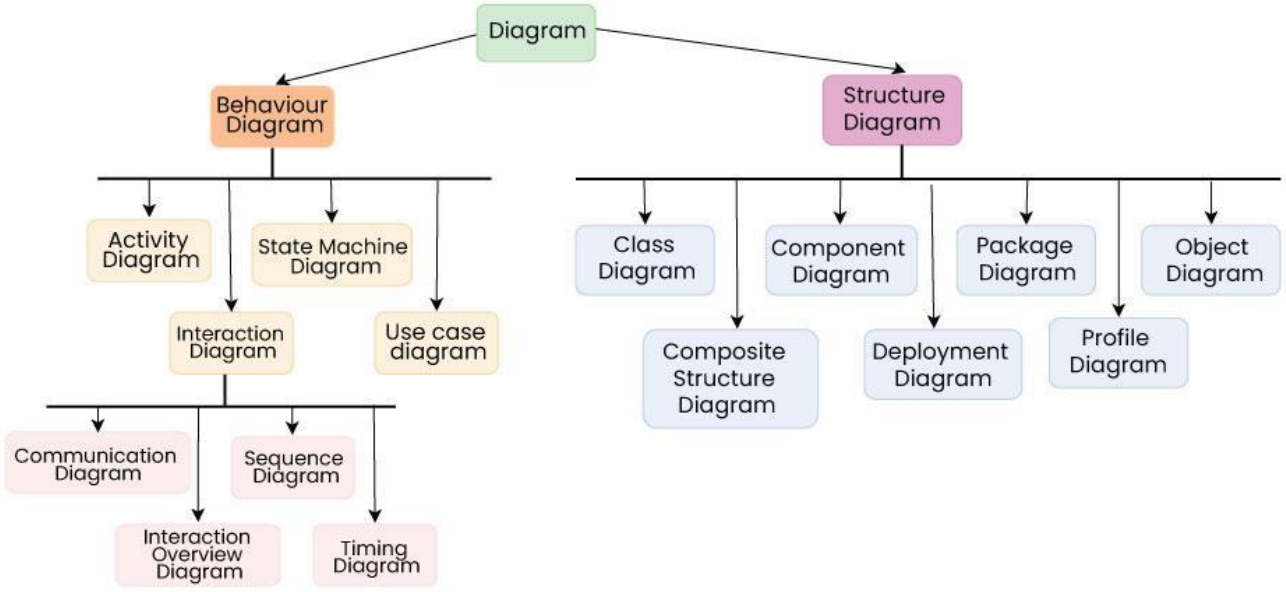
[103] - https://en.wikipedia.org/wiki/Content_delivery_network

[104] - Impact of Bottlenecks: Reduced Performance, Poor User Experience, Limited Scalability. https://en.wikipedia.org/wiki/Bottleneck_(software)

- باعث میشود ارتباطات بین افراد دخیل در پروژه، موثرتر شود، چرا که هر یک از آنها درک شفافی از نحوه عملکرد هر یک از اجزا خواهند داشت.
- باعث خواهد شد تا کد سازمان یافته تری تولید و به همان نسبت مشکلات کمتری گزارش شود.
- در نهایت سبب خواهد شد تا فرآیند کد زنی سریعتر شود چرا که برنامه نویسان، طرح ها را با جزئیات دقیقتری دنبال خواهند کرد.

طراحی سطح پائین، غالبا شامل موارد زیر است:

- Object Oriented Principles
- Process of Analyzing and Design
- UML Diagram
- Design Principles[105]
- Design Patterns[106]
- Best Practices[107]



۶) سند مفاهیم و اصول برنامه سازی (Programming Concepts/Principles) مستقل از زبانهای برنامه نویسی (Coding Syntax)

a. انتخاب طراحی و معماری مناسب پروژه
b. CPU intensive, Data Driven, DDD, Event Driven
c. انتخاب بهروشها Best Practice و الگوهای طراحی Design Patterns[۱۰۸]
d. انتخاب الگوریتمهای مناسب Algorithms برای حل مسائل
e. انتخاب ساختمان داده مناسب Data Structure برای حل مسائل
f. انتخاب امکانات پایه برنامه سازی
i. Conditions

105 - https://learn.microsoft.com/en-us/azure/architecture/guide/design-principles/
106 - https://learn.microsoft.com/en-us/azure/architecture/patterns/#pattern-catalog
107 - https://en.wikipedia.org/wiki/Best_practice
108 - https://en.wikipedia.org/wiki/Software_design_pattern , https://cio-wiki.org/wiki/Design_Pattern

ii. Iterations/Loop
iii. Arrays/Sets
iv. Functions
v. Sync(I/O Blocking)/Callbacks/Promise/Async-Await

g. انتخاب پارادایم مناسب

i. Procedural
ii. Object Oriented
iii. Functional

۷) سند نحوه انتخاب پلاتفرم، تکنولوژیها و امکانات مرتبط:

a. Cloud Computing /Native[۱۰۹]
b. Traditional Web (browser-based applications)
c. Mobile App
d. Desktop Application
e. Game
f. Embedded Systems
g. Data Science and Machine Learning
h. Blockchain
i. Artificial Intelligence (AI)

Traditional on-premises	Modern cloud[110]
▪ Monolithic	▪ Decomposed
▪ Designed for predictable scalability	▪ Designed for elastic scale
▪ Relational database	▪ Polyglot persistence (mix of storage technologies)
▪ Synchronized processing	▪ Asynchronous processing
▪ Design to avoid failures (MTBF)	▪ Design for failure (MTTR)
▪ Occasional large updates	▪ Frequent small updates
▪ Manual management	▪ Automated self-management
▪ Snowflake servers	▪ Immutable infrastructure

۸) سند طراحی بر پایه

- database first
- code first
- schema first
- model first
- Presenter first[۱۱۱]

[109] - Cloud native refers to a set of practices and technologies that let you build and run scalable applications in the cloud. https://www.cncf.io/ Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

[110] - https://learn.microsoft.com/en-us/azure/architecture/guide
https://cloud.google.com/discover

[111] - https://en.wikipedia.org/wiki/Presenter_first_(software_approach)

- Pure client-first[۱۱۲]
- Hammock Driven Development[۱۱۳]
- API[۱۱۴] first
  - Internal APIs (Private API)
  - External APIs
    - Public API[۱۱۵]
    - Open API[۱۱۶]
  - Partner APIs
  - Composite APIs
  - Unified APIs[۱۱۷]
- Container-first and cloud-native[۱۱۸]
  - PaaS (platform as a service)[۱۱۹]
  - IaaS (infrastructure as a service)
  - SaaS (software as a service)
  - CaaS (containers as a service)
- AI-First[۱۲۰]

۹) سند معیارهای انتخاب تکنولوژی ساخت

1) Ability to have separated modules to split logic
2) Interoperability, modules could extend one another
3) Inversion of Control / Dependency Injection
4) Type-safety everywhere
5) Asynchronous & Type-safe Event Management

[112] - These apps can be deployed to a CDN or static hosting service without a server (SPA, CSR, SSG), https://learning-notes.mistermicheels.com/processes-techniques/client-first-design/

[113] - https://learning-notes.mistermicheels.com/mindset/hammock-driven-development/ , https://www.youtube.com/watch?v=f84n5oFoZBc

[114] - https://en.wikipedia.org/wiki/API

APIs (Application Programming Interfaces) can be categorized based on their access, purpose, and structure:

- **Based on Access and Purpose:**
  - **Internal APIs (or Private APIs):** These APIs are designed for use exclusively within an organization. They facilitate communication and data exchange between different internal systems, applications, or departments, enhancing productivity and streamlining internal processes. They are not exposed to external developers or users. (User authentication, Data retrieval, Workflow automation, Notification/alerting, …)
  - **External APIs (or Open APIs or Public APIs):** These APIs are made publicly available for external developers and third-party applications to integrate with a company's services or data. They promote collaboration and enable the creation of new applications and functionalities by leveraging existing services. (sign in with google, Google map, Waze, Weather APIs, Payment gateway, Social media, YouTube videos)
  - **Partner APIs:** These APIs are shared with specific, trusted business partners to facilitate secure and controlled data exchange and integration between organizations. Access is typically restricted and managed through agreements and security measures like API keys or OAuth. (payment processing or data syndication integration, Fast Healthcare Interoperability Resources (FHIR®))
- **Based on Structure and Functionality:**
  - **Composite APIs:** These APIs combine multiple individual API requests into a single, unified call. This approach can improve performance and efficiency by reducing the number of round trips between the client and server, especially in scenarios where a single operation requires interactions with multiple backend services. They are particularly useful in microservices architectures where different functionalities might reside in separate services.

[115] - **Open APIs** are entirely free and entirely open to the public.

[116] - **Public APIs** are similar to open APIs, but they tend to have a price tag.

[117] - **Unified APIs** are similar to composite APIs, except they bundle together multiple APIs and backend resources into one.

[118] - https://cloud.google.com/learn/what-is-cloud-native

[119] - https://cloud.google.com/learn/paas-vs-iaas-vs-saas

[120] - The AI-first stack is designed for applications that heavily rely on artificial intelligence and machine learning. It typically includes technologies like Python, LangChain, OpenAI APIs, and vector databases such as Pinecone or FAISS.

6) Easy integration with DB Technology (incl. data validation)
7) Isomorphism to have the same concepts in Node, Deno, Browser, ...
8) **Real-time data solutions (vs content-centric projects)**
    - Apps that require high-frequency updates from the server:
        - Gaming, Social networks, Voting, Auctions, GPS apps
    - Dashboards and monitoring apps:
        - Company dashboards, Live maps, Instant sales updates, Travel alerts
        - Continuous integration/continuous delivery (CI/CD) pipeline pages
    - Collaborative and multi-user interactive apps:
        - Whiteboard apps, Team meeting apps, Document sharing apps
        - Visual Studio Live Share
    - Apps that require instant notifications:
        - Email apps, Chat apps, Turn-based games, Time series reporting
        - GitHub Actions, issue and pull request systems

۱۰) سند انتخاب پارادایم برنامه سازی

("Peter Van Roy (2009-05-12). "Programming Paradigms: What Every Programmer Should Know[۱۲۱] رجوع شود به)

۱۱) Prototyping

a. Rapid Throwaway Prototyping
b. Evolutionary Prototyping
c. Incremental Prototyping
d. Extreme Prototyping

۱۲) انتخاب نوع توسعه (مونولیتیک، ماجولار، هسته+پلاگین)

۱۳) سند معیارهای انتخاب سکوی برنامه سازی [۱۲۲]:

a. Programming Languages Type/Categories [123]

- **Declarative languages**
- Fourth-generation languages
- **Functional Programming** (Haskell, F#)
- Imperative languages
- Interpreted languages
- Iterative languages
- **Multiparadigm languages**
- **Object-oriented class-based languages** (C++, Java, C#)
- Object-oriented prototype-based languages
- **Procedural languages**
- Reflective languages
- Rule-based languages
- **Scripting languages**
- **System languages**

[121] - https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf

[122] - Before starting a new project, there is a difficult decision on whether it will be based on a framework or on a CMS. When choosing to use a framework, you need to spend much time creating CMS features for the project. On the other hand, when choosing to use a CMS, it's more difficult to build custom application functionality. It is impossible or at least very hard to customize the core parts of the CMS.
https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2023/
https://w3techs.com/technologies/overview/programming_language

[123] - https://en.wikipedia.org/wiki/List_of_programming_languages_by_type , https://en.wikipedia.org/wiki/Timeline_of_programming_languages , https://programming-language-benchmarks.vercel.app/ , https://en.wikipedia.org/wiki/The_Computer_Language_Benchmarks_Game , https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html , https://en.wikipedia.org/wiki/Fifth-generation_programming_language

b. Parallelism Model:[124] (Shared Memory, Message passing (Asynchronous/Synchronous)[125])

- **Event Loop**[126] (JavaScript (Node.js[127], Deno[128], Bun[129]))
- **Multi-threading**[130] (C++, Java, C#)
- **CSP-based**[131] (Occam, Crystal, Go[132])
- **Actor model**[133] (Erlang/Elixir/LFE/Gleam, Scala/Akka[134], D, Dart, Ruby, Haskell, Swift, Rust,…)

c. Frameworks[135] and Toolkits

- Actix-web, Axum, …
- GoFiber, Phoenix
- ExpressJS, NestJS, Ultimate-express, Fastify, Hapi, …
- Django, FastAPI, Flask
- **Azure Service Fabric, Actix, go-actor, Actr, …**
  - Erlang/Elixir/LFE/Gleam[136]
  - .Net (**Orleans, Dapr, Akka.NET, Proto.Actor**) [137]
  - JVM (**AKKA,** Apache Pekko, ReactorsIO, Vert.x, VLINGO XOOM)[138]
  - C++ (**CAF, Sobjectizer**, Actor-CPP) [139]
  - Python (Pykka, Plusar, ray)
  - Typescript (Tarant)
  - …

d. Platforms/Ecosystem (Cloud, Web, Desktop, Mobile, Game, IoT, Machine Learning, API, Microservice, …)

---

[124] - https://en.wikipedia.org/wiki/List_of_concurrent_and_parallel_programming_languages , https://www.openmp.org/ , https://pkolaczk.github.io/memory-consumption-of-async/

[125] - https://en.wikipedia.org/wiki/Message_Passing_Interface
https://www.mpi-forum.org/ , https://www.open-mpi.org/ , https://www.mpich.org/ , Intel® MPI Library
https://developer.nvidia.com/mpi-solutions-gpus , https://developer.nvidia.com/mvapich , https://developer.nvidia.com/ibm-spectrum-mpi

[126] - Event Loop is a single-threaded concurrency model that manages asynchronous operations using a queue and callbacks, suitable for I/O-bound tasks but can become a bottleneck for CPU-bound operations. Event Loop is a single-threaded model focused on efficient handling of asynchronous I/O operations. Non-blocking, allowing the main thread to handle other tasks while waiting for asynchronous operations to complete.
https://en.wikipedia.org/wiki/Event_loop , https://en.wikipedia.org/wiki/Loop-level_parallelism

[127] - https://nodejs.org

[128] - https://deno.com

[129] - https://bun.sh

[130] - https://en.wikipedia.org/wiki/Multithreading_(computer_architecture) , https://en.wikipedia.org/wiki/Thread_(computing)

[131] - https://en.wikipedia.org/wiki/Communicating_sequential_processes

[132] - **Do not communicate by sharing memory; instead, share memory by communicating.** https://go.dev/doc/effective_go#concurrency

[133] - https://en.wikipedia.org/wiki/Actor_model , https://en.wikipedia.org/wiki/History_of_the_Actor_model
Actor Model can be used to solve the following types of problems:

- Concurrency
- Stream Processing
- Event-Driven Programming
- Event Sourcing and CQRS
- Location Transparency
- Highly Available, Fault-Tolerant Distributed Systems
- Low Latency, High Throughput

[134] - https://en.wikipedia.org/wiki/Akka_(toolkit) , https://akka.io/

[135] - همواره از فریم ورک ها و Stack های برتر و Type Safe استفاده شود، اگر برنامه نویس و یا پیمانکاری ادعا کرد که از ابزار و یا فریم ورکی که خود توسعه داده است استفاده میکند، بدون تردید آنرا کنار بگذارید. برای مقایسه کارائی فریم ورکها به منابع زیر رجوع شود :
https://www.techempower.com/benchmarks ,https://web-frameworks-benchmark.netlify.app/result ,
https://hotframeworks.com/ , https://2024.stateofjs.com/en-US/libraries/#tier_list ,
https://survey.stackoverflow.co/2025/technology , https://krausest.github.io/js-framework-benchmark/current.html
https://www.tiobe.com/tiobe-index/ , **https://innovationgraph.github.com/global-metrics/programming-languages** , https://redmonk.com/data/

[136] - http://cloudi.org/

[137] - https://dapr.io/ , https://proto.actor/ , https://github.com/asynkron , https://getakka.net/ , https://github.com/akkadotnet/akka.net

[138] - https://akka.io/ , https://pekko.apache.org/ , https://vlingo.io/ , http://reactors.io/ , https://vertx.io/

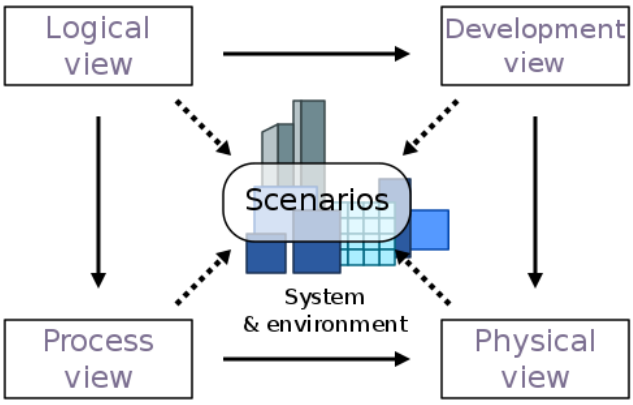
[139] - https://www.actor-framework.org/ , https://github.com/Stiffstream/sobjectizer

- Java (Quarkus (Vert.x/Netty/NIO), Micronaut, Helidon, Spring Boot Webflux (Netty/NIO), …)
- .Net

e. CMS
- Wordpress, Joomla, Drupal, DotNetNuke, …

۱۴) سند طراحی[۱۴۰] و معماری[۱۴۱] ۴+۱

مدل ۴+۱ توسط فیلیپ کروتچن برای "توصیف معماری سیستم‌های نرم‌افزاری" معرفی شد. این مدل مبتنی بر استفاده از چند view است. viewها برای توصیف سیستم از دید مصرف کنندگان مختلف و سرمایه‌گذاران نرم‌افزار است مانند کاربران نهایی، برنامه نویسان نرم‌افزار و مدیران پروژه. ۴ view در این مدل شامل **مدل منطقی، توسعه، فرایند و فیزیکی** می‌شود. هر یک از دیدگاه های این معماری با یکی یا چند نمودار uml در ارتباط است[۱۴۲].



a. **مدل منطقی** Logical view : این دیدگاه نشان میدهد که عملکرد سیستم چگونه توسط طراحی داخلی فراهم میشود این دیدگاه ساختار ایستا و پویای داخل سیستم را مشخص میکند از جمله نمودار هایی با این دیدگاه در ارتباط اند می توان به **Class diagrams & Object diagrams** اشاره کرد.

b. **مدل توسعه** Development(implemention) View : دید توسعه برای تشریح سیستم از دید یک برنامه‌نویس است و درگیر مدیریت نرم‌افزار است. به این View همچنین Implementation View هم می گویند. در UML از نمودار **Component diagrams** برای توصیف اجزای سیستم استفاده می‌کند.

c. **مدل فرآیند** Process View : دید فرایند، ، درگیر وجهه پویای سیستم است. این دیدگاه المان های سیستم و ارتباط آن ها با هم و ترتیب انجام کارها بر اساس زمان را بررسی میکند از جمله نمودار های با این بخش در گیرند میتوان به **Sequence diagrams, Activity diagrams, State machine diagram** اشاره کرد.

d. **مدل فیزیکی** Physical(deployment) View : دید فیزیکی سیستم را از دید یک مهندس سیستم نمایش می‌دهد. این دید درگیر توپولوژی کامپونت‌های نرم‌افزاری در لایه فیزیکی است، به‌علاوه ارتباطات فیزیکی بین این اجزا. در UML از نمودارهای **Deployment diagrams** برای نمایش لایه فیزیکی استفاده می‌شود.

e. **سناریوها** Use Case : این دیدگاه دید کابران خارجی نسبت به نرم افزار را مورد بررسی قرار میدهد نمودار که در uml این دیدگاه را پوشش میدهد **Use case diagram** است.

۱۵) سند طراحی تفضیلی

a. Function Oriented[۱۴۳] Design

---

140 - https://en.wikipedia.org/wiki/C4_model , https://c4model.com/

141 - https://en.wikipedia.org/wiki/4+1_architectural_view_model

142 - https://www.omg.org/spec/category/software-engineering/
https://www.omg.org/spec/category/modeling/ , https://www.omg.org/spec/SysML/1.6/PDF
https://www.omg.org/spec/category/business-modeling/

143 - Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.

b. Object-Oriented[۱۴۴] Design

۱۶) Data Validation

۱۷) هماهنگی/تجمیع/انسجام[۱۴۵] با سایر سیستمهای سازمان

۱۸) Lean Software Development[۱۴۶]

۱۹) Business Analysis

۲۰) System Analysis

۲۱) استراتژی Top-Down و یا Bottom-up[۱۴۷]

۲۲) سند معیارهای انتخاب نوع پایگاه داده SQL/NoSQL

a. Relational
b. Object Relational
c. Multi-Dimensional
d. NoSQL
   i. Key–value store
   ii. Document store
   iii. Graph
   iv. Object database
   v. Tabular
   vi. Tuple store
   vii. Triple/quad store (RDF) database
   viii. Hosted
   ix. Multivalue databases
   x. Multimodel database
   xi. Wide Column Store
   xii. Native multi-model database

۲۳) انتخاب DB Engine[۱۴۸]

۲۴) بررسی نقشه راه[۱۴۹] مطلوب

۲۵) سند مرزها و محدودیتها[۱۵۰]

a. **Technical Limitations**
   i. Limited scalability
   ii. High costs
   iii. Lack of flexibility
   iv. Lack of privacy
   v. The security model
   vi. Critical size
   vii. Hidden centrality
b. **Nontechnical Limitations**
   i. Lack of legal acceptance
   ii. Lack of user acceptance

144 - In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data.

145 – Integration

146 - https://en.wikipedia.org/wiki/Lean_software_development حذف اتلاف منابع، تمرکز بر ارزش مشتری، و بهینه‌سازی جریان توسعه نرم‌افزار.

147 - https://en.wikipedia.org/wiki/Bottom-up_and_top-down_approaches

148 - https://en.wikipedia.org/wiki/Database , https://en.wikipedia.org/wiki/Category:Types_of_databases https://en.wikipedia.org/wiki/ACID ,

149 - https://roadmap.sh/

150 - https://en.wikipedia.org/wiki/C10k_problem

۲۶) SDK[۱۵۱]

۲۷) انتخاب Rule Engine

۲۸) انتخاب Workflow Engine[۱۵۲]

a. Workflow as DSL (Domain-Specific Language)

b. Workflow as Code

۲۹) انتخاب Message Broker[۱۵۳]

۳۰) API Architecture/Protocols **(Richardson Maturity Model 2008)**[۱۵۴]

a. GraphQL[۱۵۵]

b. REST[۱۵۶]

c. SOAP[۱۵۷]

d. XML RPC[۱۵۸], Protobuf/gRPC[۱۵۹]

e. Webhooks[۱۶۱], WebSocket[۱۶۲], Server-Sent Events (SSE)[۱۶۰]

---

151 - Software Development Kit: Toolbox to build applications

152 - https://github.com/meirwah/awesome-workflow-engines , https://devblogs.microsoft.com/ise/guide-to-workflow-engines/

153 - Focus on the following five characteristics:

- **Message queuing model:** How are messages passed between two parties? Is it via a stream or a queue?
- **Delivery guarantee:** Are messages always delivered at least once, or is this not always the case?
- **Ordering guarantee:** Are messages delivered in the order they were sent, or are they not?
- **Throughput and latency:** How many messages can the platform handle, and how fast is the communication? Keep in mind that all these systems can scale to handle increased throughput, and that results will vary based on your system configuration.
- **Persistence and replayability:** Does the platform store messages and allow for reprocessing if they were missed the first time?

Note: Kafka the exact-once delivery guarantee is a more advanced feature that can be achieved through the use of idempotent producer and transaction APIs. Exactly-once delivery provides a higher level of reliability, but it requires a more complex setup and careful management to ensure messages are not processed multiple times in the event of transaction failures.

Communication through the exchange of messages using Message brokers is carried out based on **two distinct messaging patterns** or styles. They are known as **Point-to-point messaging and Publish/Subscribe messaging**:

- **Point-to-point messaging:** This communication model is the distribution pattern used in message queues where the sender and recipient of each message are associated on a one-to-one basis. Every message in the queue is read only once and only sent to one recipient. Point-to-point messaging model is implemented in Payroll management and Financial Transaction Processing scenarios where assurance needs to be provided that each payment is made only once. If the consumer is offline, the message broker stores it in the message queue and delivers it at a later time.
- **Publish/Subscribe messaging:** In this mode of message exchange, the producer is completely unaware of who will be the consumer of the message. It sends messages concerning a topic, and all applications that have subscribed to it receive all published messages. The consumer and producer have a one-to-many relationship, and the model is used in the event-driven architecture-based system, where applications have no dependencies on each other.

https://en.wikipedia.org/wiki/Message_broker , https://en.wikipedia.org/wiki/Message-oriented_middleware , https://en.wikipedia.org/wiki/Enterprise_messaging_system , https://en.wikipedia.org/wiki/Flow-based_programming , https://en.wikipedia.org/wiki/Comparison_of_business_integration_software , https://en.wikipedia.org/wiki/Enterprise_service_bus , https://www.enterpriseintegrationpatterns.com/ , https://ultimate-comparisons.github.io/ultimate-message-broker-comparison/

154 - https://en.wikipedia.org/wiki/Richardson_Maturity_Model

155 - https://graphql.org/

156 - https://restfulapi.net/ , https://en.wikipedia.org/wiki/REST , https://en.wikipedia.org/wiki/Overview_of_RESTful_API_Description_Languages

157 - https://www.w3.org/TR/soap/

158 - https://xmlrpc.com/

159 - https://grpc.io/

160 - https://github.com/wso2/reference-architecture/blob/master/event-driven-api-architecture.md

161 - https://en.wikipedia.org/wiki/Webhook , https://webhook.net/ , https://docs.github.com/en/webhooks/about-webhooks

162 - https://en.wikipedia.org/wiki/WebSocket , https://en.wikipedia.org/wiki/Comparison_of_WebSocket_implementations , https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API , https://en.wikipedia.org/wiki/BOSH_(protocol)

f. WebRTC[۱۶۳] (Web Real-Time Communication)

g. JSON, TOON (Token-Oriented Object Notation)[۱۶۴]

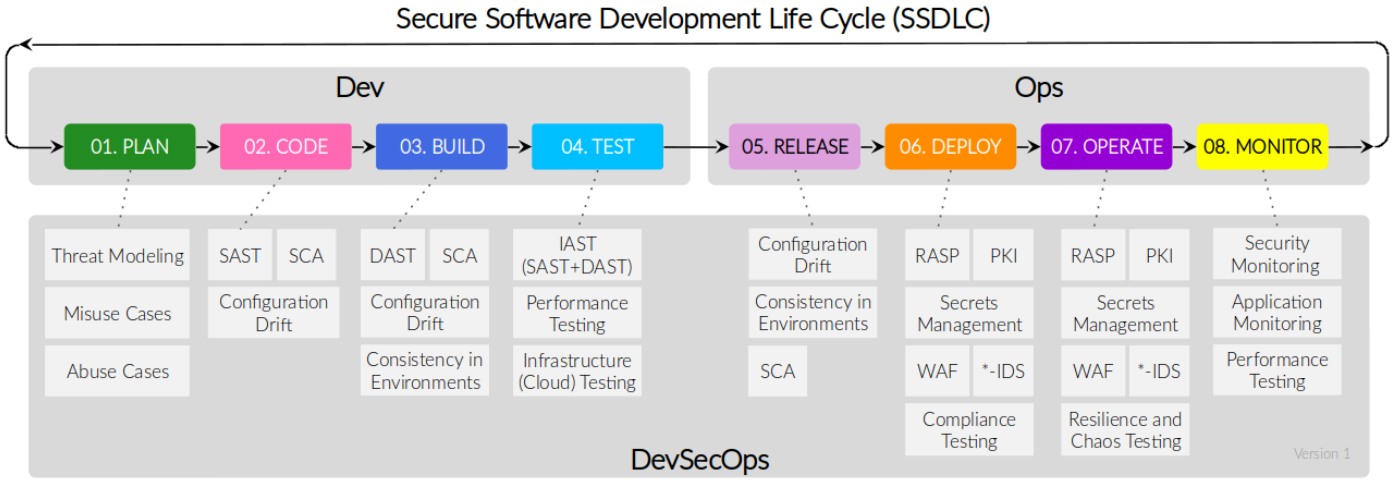
۳۱) مکانیزم دریافت/ارسال داده از/به سایر سیستم ها OLTP/ETL/OLAP

۳۲) لایه های نرم افزار:

a. Application layer
b. Execution layer
c. Semantic layer
d. Propagation layer
e. Leader election and Consensus Protocol layer

۳۳) سند توصیف استانداردهای متدولوژی[۱۶۵]

a. نقشه راه استراتژیک و برنامه انتشار
b. مدل آبشاری Waterfall Model
c. مدل تدریجی Incremental Model
d. مدل حلزونی/مارپیچ Spiral Model
e. RAD
f. مدل چابک[۱۶۶] Agile
i. Extreme Programming
ii. Kanban
iii. Scrum
g. DevSecOps[۱۶۷]



h. CI/CD[۱۶۸] Code Pipeline
i. continuous development

163 - https://webrtc.org/ , https://en.wikipedia.org/wiki/WebRTC
164 - https://github.com/toon-format/toon , https://github.com/toon-format/spec/blob/main/SPEC.md , https://github.com/toon-format/spec/tree/main/examples , https://jsontoon.net/
165 - https://en.wikipedia.org/wiki/Software_development_process
166 - https://www.agilealliance.org , https://agile2.net/ , https://agilemanifesto.org/ , https://en.wikipedia.org/wiki/Agile_software_development
167 - https://holisticsecurity.io/2020/02/10/security-along-the-sdlc-for-cloud-native-apps
168 - https://en.wikipedia.org/wiki/CI/CD

ii. continuous testing
   ۱. Stability
   ۲. Scalability
   ۳. Speed
   ۴. Security
iii. **continuous integration**[۱۶۹]
iv. **continuous delivery**[۱۷۰]
v. **continuous deployment**
vi. continuous monitoring

i. SAFe[۱۷۱]
j. LeSS[۱۷۲]
k. Big bang model: بر منابع تاکید داشته و برنامه ریزی چندانی نیاز نیست. نیازمندیها به محض وصول، درک شده و اجرا می شوند. مناسب پروژه های کوچک و آکادمیک.
l. Spec-Driven Development[۱۷۳] (SDD) (به پیوست ۸ مراجعه شود)

نقطه تعادل و بالانس مناسب بین اولویتها Tradeoffs[۱۷۴] and Priorities

برای چگونگی پروسه نگاه کنید به پیوست ۷. و همچنین[۱۷۵] Software Non-functional Assessment Process, SNAP Points.

- Innovation vs Risks vs Opportunity vs Threat
- Consistency vs. Availability **(CAP Theorem)**
- Single Point of Failure and Bottlenecks
- **Push** (Event-Driven (Event-Based, Asynchronous communication), Reactive Streaming, publish-subscribe model(messages to multiple queues & consumers)) vs **Pull** (Message Driven (Request-Based, Synchronous requests and responses, Point-to-Point communication)) Architecture
- Designing Scalable Data Pipelines: Batch, **Streaming**[176] and Layered Architectures
  - Lambda architecture

[169] - https://en.wikipedia.org/wiki/Continuous_integration
[170] - https://en.wikipedia.org/wiki/Continuous_delivery
[171] - Scaled Agile Framework, https://en.wikipedia.org/wiki/Scaled_agile_framework
[172] - Large-Scale Scrum, https://en.wikipedia.org/wiki/Scrum_(software_development)#Large-scale_scrum
[173] - What is a spec? the closest I've seen to a consistent definition is the comparison of a spec to a "Product Requirements Document". A spec is a structured, behavior-oriented artifact - or a set of related artifacts - written in natural language that expresses software functionality and serves as guidance to AI coding agents. Each variant of spec-driven development defines their approach to a spec's structure, level of detail, and how these artifacts are organized within a project.
Instead of coding first and writing docs later, in spec-driven development, you start with a spec. Spec-driven development means writing a "spec" before writing code with AI ("documentation first"). The spec becomes the source of truth for the human and the AI. SDD approaches and definitions:
- **Spec-first:** A well thought-out spec is written first, and then used in the AI-assisted development workflow for the task at hand.
- **Spec-anchored:** The spec is kept even after the task is complete, to continue using it for evolution and maintenance of the respective feature.
- **Spec-as-source:** The spec is the main source file over time, and only the spec is edited by the human, the human never touches the code.

[174] - Everything in software architecture is a trade-off.
[175] - https://en.wikipedia.org/wiki/SNAP_Points
[176] - Streaming Data (Bounded and Unbounded) is a log of events.
See this: https://softwareengineeringdaily.com/2016/08/19/apache-beam-with-frances-perry/
https://learn.microsoft.com/en-us/azure/architecture/databases/guide/big-data-architectures
https://medium.com/@georgemichaeldagogomaynard/understanding-different-data-pipeline-architectures-lambda-kappa-medallion-and-delta-5feb35a32c38
https://medium.com/@c.chai/main-architecture-types-in-data-engineering-5e2242cace15

    - Kappa architecture
    - Delta architecture
    - Medallion architecture
- Balance between **Part 1**, **Part 2, Part 3** and **Part 4** :
    - **Part 1:** Scalability vs Productivity vs Complexity
    - **Part 2:** Throughput (Volume of Tasks/Requests/Operations) vs Latency (Time Delay)[177]
    - **Part 3:** Team size vs Product Value vs Cost vs Safety & Reliability
    - **Part 4:** Accuracy[178] (vs/and) Precision[179] vs Performance
- Communications (Synchronous vs Asynchronous)
    - Synchronous calls: REST, GraphQL, WebSocket, Webhooks, RPC/gRPC, SOAP
    - Asynchronous messaging: Kafka, RabbitMQ, NATS, …
- Stateful vs. Stateless Architecture
- Immediate Consistency vs. Eventual Consistency Strong vs Eventual Consistency[180]
- Modular Monolith vs. Microservices Architecture
    - **Modular Monolith Architecture**
        i. Advantage
            1. Low complexity
            2. Easy to deploy
        ii. Disadvantage
            1. Rigid
            2. Hard to scale
            3. Slow performance
            4. Single point of failure
            5. Slow continous develpment
        iii. Other
            1. same data center same server
    - **Microservices Architecture**
        i. Advantage
            1. Loosely Coupled
            2. Agile & Flexible
            3. Independent development
            4. Fault isolation
            5. Mixed technology stack
            6. Granular scaling
        ii. Disadvantage
            1. High complexity
            2. Data Consistency
            3. Automation & Response time
            4. Debugging
        iii. Other
            1. same data center dif server
            2. same region dif data center
            3. same country dif region
            4. dif country
- HTTP Polling/Streaming (Long/Short) vs SSE vs WebSocket vs Webhooks **(Real time communication Technologies)**

[177] - in **Ultra-Low-Latency Systems** *(for example High Frequency Trading **HFT**)* Reducing latency is a never-ending battle. Every microsecond count, and even small improvements can have a big impact.
[178] - **Accuracy:** Ensure operations are executed correctly and logged without errors.
[179] - https://labwrite.ncsu.edu//Experimental%20Design/accuracyprecision.htm
[180] - Match Consistency to user expectations.

- XML vs JSON vs TOON[181]
- Synchronous vs. Asynchronous Processing
- Vertical vs. Horizontal Scaling
- Single Tenant vs Multi Tenant API (SaaS Architecture Maturity Model)
- Flexibility vs. Simplicity
- Control vs. Convenience
- Local-first architecture vs Global-first architecture vs Offline-first architecture
- Short-Term vs. Long-Term Goals
- Data Normalization vs. Denormalization
- Relational (SQL) vs. Non-relational (NoSQL) Databases
- Read vs. Write Optimization
- Serverless vs Serverful
- Tech Stacks (LAMP vs MEAN vs MERN vs JAMstack vs …)
- API vs MCP[182] (Model Context Protocol) vs A2A (Agent-to-Agent communication layer)
- tcp vs SSL vs TLS vs ipc vs nats
- Frameworks vs Libraries (Inheritance vs Composition)
- CLI vs UI
- Waterfall/SDLC vs Spiral vs Agile vs SDD (Spec-Driven Development)
- Database vs Blockchain[183]
- Memory safety & Garbage Collection[184]
- Compiler vs Interpreter vs Transpiler[185]
- HTTP/1.1**(TCP)** vs HTTP/2**(TCP+Stream)** vs HTTP/3**(UDP) (QUIC: Quick UDP Internet Connections)** [186]
- System software vs Application software vs Embedded software
- Zonal Architecture vs. Domain Architecture (Cloud Deployment)
- Data Serialization Standards & Format (Binary, JSON/XML, Schema-Based)
- Frontend
  - MVC (+Template Engines)[187] vs Backends for frontends vs Microfrontends
  - SSR vs CSR vs SSG vs ISR vs SPA[188]
- Three 9s vs Four 9s Availability (99.9% vs 99.99%)

---

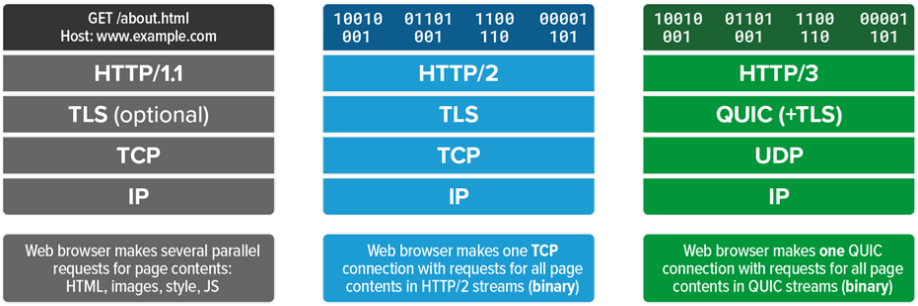
[181] - **Use Cases: Where TOON Shines**
- LLM Prompt Compression
- Chatbots with Tabular Inputs
- Multi-step AI Agents
- RAG / AI Document Readers
- Data Summarization
- Prompt Dataset Generation

When Not to Use TOON
- Deeply nested or non-uniform structures (tabular eligibility ≈ 0%): JSON-compact often uses fewer tokens. Example: complex configuration objects with many nested levels.
- Semi-uniform arrays (~40–60% tabular eligibility): Token savings diminish. Prefer JSON if your pipelines already rely on it.
- Pure tabular data: CSV is smaller than TOON for flat tables. TOON adds minimal overhead (~5-10%) to provide structure (array length declarations, field headers, delimiter scoping) that improves LLM reliability.
- Latency-critical applications: If end-to-end response time is your top priority, benchmark on your exact setup. Some deployments (especially local/quantized models like Ollama) may process compact JSON faster despite TOON's lower token count. Measure TTFT, tokens/sec, and total time for both formats and use whichever is faster
- Use in production APIs or storage formats

[182] - https://en.wikipedia.org/wiki/Model_Context_Protocol , https://modelcontextprotocol.io/
[183] - https://www.geeksforgeeks.org/dbms/difference-between-blockchain-and-a-database/
[184] - https://en.wikipedia.org/wiki/Memory_safety , https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
[185] - https://en.wikipedia.org/wiki/Compiler , https://en.wikipedia.org/wiki/Interpreter_(computing) , https://en.wikipedia.org/wiki/Source-to-source_compiler , https://en.wikipedia.org/wiki/WebAssembly , https://webassembly.org/
[186] - https://en.wikipedia.org/wiki/HTTP , https://en.wikipedia.org/wiki/HTTP/2 , https://en.wikipedia.org/wiki/HTTP/3 , https://en.wikipedia.org/wiki/QUIC
[187] - https://www.cs.usfca.edu/~parrt/papers/mvc.templates.pdf
[188] - Server-side rendering, Client-side rendering, Static site generator, Incremental Static Regeneration, Single-page application https://en.wikipedia.org/wiki/Static_site_generator , https://en.wikipedia.org/wiki/Single-page_application

- Layer 4 vs Layer 7 Load balancing
- Policy vs Details
- …

بدیهی است که پس از انتخاب هریک از گزینه های فوق، بدهی فنی شناخته شده (و هدفمندی) نیز در بدنه سیستم و محصول نهائی شکل خواهد گرفت.



طراحی پایگاه داده[189]

۱) ابزار طراحی مدل ER
۲) طراحی 3NF
۳) لیست جداول
۴) لیست ستونها/فیلدها
۵) کاربرد هر یک از جداول و ستونها
۶) ایندکسها (و انواع آن) و PK
   a. روش به روزآوری ایندکسها
   b. فواصل زمانی به روز آوری ایندکسها
   c. ارائه ایندیکتوری که تریگر فعال سازی به روز آوری ایندکسها را فعال کند.
۷) جامعیت ارجاعی و FK
۸) لیست Viewها و نحوه عملکرد آن
۹) لیست Procedureها و نحوه عملکرد آن
۱۰) لیست Triggerها و نحوه عملکرد آن
۱۱) لیست Functionها و نحوه عملکرد آن
۱۲) لیست کاربران
۱۳) لیست مجوزها و نحوه تخصیص آنها
۱۴) Character set & Collation
۱۵) ORM
   a. Data mapper pattern
   b. Active record pattern
   c. Object–relational mapping

[189] - Database, Data Warehouse, Data Lake

فاز ۳) چرخه توسعه نرم افزار[۱۹۰]

۱) محل پیاده سازی منطق کسب وکار Business Logic:

a. MVC(Model)

b. Database (Stored Procedure)

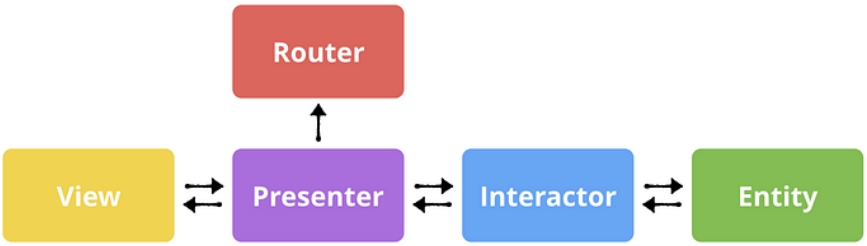
۲) مستندات فریم ورکها، میکروفریم ورک، دلیل انتخاب و نحوه مدیریت و به روز آوری آن

a. Backend

- Synchronous I/O (PHP, Python, …)
- Asynchronous I/O (NodeJS, reactPHP, …)

b. Frontend[۱۹۱]

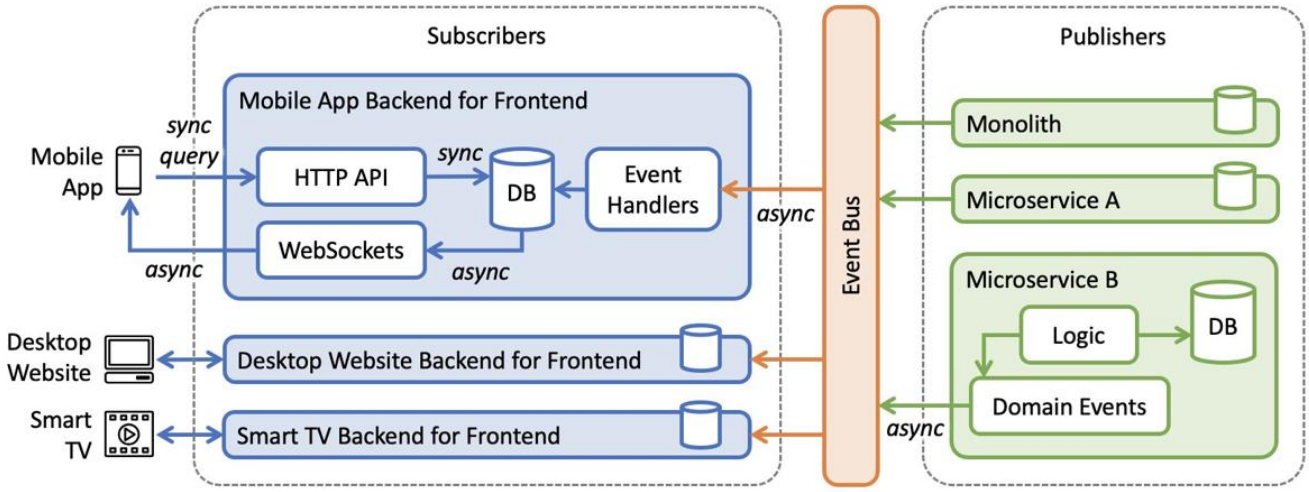
- معماری
  ۱. MVC[۱۹۲]
  ۲. HMVC[۱۹۳]
  ۳. Model-View-ViewModel (MVVM)[۱۹۴]
  ۴. MVVM-C(Coordinator)
  ۵. MVP[۱۹۵]
  ۶. MVA[۱۹۶]
  ۷. Model-View-Intent (MVI)[۱۹۷]
  ۸. MVT (Model-View-Template)
  ۹. PAC[۱۹۸]
  ۱۰. **VIPER** (View, Interactor, Presenter, Entity, Router)



- Patterns
  ۱. Static Pages
  ۲. MVC
  ۳. SPA

190 - https://en.wikipedia.org/wiki/Systems_development_life_cycle , https://en.wikipedia.org/wiki/List_of_software_development_philosophies , https://cio-wiki.org/wiki/Software_Development_Life_Cycle_(SDLC)
https://en.wikipedia.org/wiki/Application_lifecycle_management
191 - https://en.wikipedia.org/wiki/Front_end_and_back_end , https://en.wikipedia.org/wiki/Front-end_web_development , https://en.wikipedia.org/wiki/Htmx
192 - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
193 - https://en.wikipedia.org/wiki/Hierarchical_model%E2%80%93view%E2%80%93controller
194 - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel
195 - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter
196 - https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93adapter
197 - https://www.geeksforgeeks.org/software-engineering/model-view-intent-mvi-pattern-in-reactive-programming-a-comprehensive-overview/
198 - https://en.wikipedia.org/wiki/Presentation%E2%80%93abstraction%E2%80%93control

۴. Modular Frontend Monolith
۵. Backends for frontends[۱۹۹]
۶. Microfrontends[۲۰۰]



- Responsive واکنشگرا
- Progressive پیش رونده
- Reactive programming[۲۰۱] برنامه نویسی واکنشی
- Server side & Client-side Template engine
- CSS Framework (Bootstrap, Tailwind CSS, …)
- JavaScript Framework (React, Angular, VUE, …)

Fullstack .c

۳) مستندات اپلیکیشن، نحوه مدیریت و به روز آوری آن
۴) مستندات تطبیق فرآیندها و نیازمندیها با ماجولهای طراحی شده
۵) ساختار فایلها و دایرکتوری های ثابت و نقش هریک
   a. *.php, *.html, *.css, *.js
   b. *.jpg, *.png, *. webp
   c. …
۶) ساختار فایلها و دایرکتوری هایی که رشد می یابند و نقش هریک
   a. *. uploads
   b. *. emails
   c. *. logs
   d. …

[199] - Backend-for-Frontend is an architecture that proposes creating dedicated backend layers for each frontend client. In traditional architectures, all clients interact with a single API. However, since each client has distinct requirements, this approach can lead to complications. BFF addresses these issues by creating separate backends for each client type.
https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends ,
https://www.geeksforgeeks.org/system-design/backend-for-frontend-pattern/ , https://reactrouter.com/explanation/backend-for-frontend ,
https://aws.amazon.com/blogs/mobile/backends-for-frontends-pattern/
[200] - https://en.wikipedia.org/wiki/Micro_frontend , https://micro-frontends.org/ , https://martinfowler.com/articles/micro-frontends.html
[201] - https://en.wikipedia.org/wiki/Reactive_programming

۷) ساختار فایلها و دایرکتوری های موقت و نقش هریک

۸) مستندات Best Practice[۲۰۲] و استانداردهای[۲۰۳] مورد استفاده

a. File Encoding
   - Unicode/UTF-8
   - ASCII

b. Autoload Standards[۲۰۴](PSR-0, **PSR-4**)

c. Programming Style
   - Simplicity
   - Control Constructs
   - Nesting
   - Information Hiding
   - User-Defined Types
   - Side Effects
   - Module size
   - Module Interface
   - Name Space

d. Coding Standards[۲۰۵]
   - Code Well documented
   - Inline comments
   - Line Length
   - Spacing
   - Indentation
   - Use of global
   - Naming conventions
   - Error Messages & Excention handling
   - Structured Programming[۲۰۶] Rules
   - Pseudo Codes
   - Don’t use goto

e. HTML Forms Implementation

۹) تطبیق استانداردهای سورس کد با آنچه که پیاده سازی شده است

۱۰) سند بهینه سازی کدها[۲۰۷]

202 - http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

203 - https://en.wikipedia.org/wiki/Software_standard , https://en.wikipedia.org/wiki/World_Wide_Web_Consortium , https://en.wikipedia.org/wiki/Internet_Society , https://en.wikipedia.org/wiki/Open_standard , https://en.wikipedia.org/wiki/Electronic_data_interchange , https://users.cecs.anu.edu.au/~kambara/old_site/software/software_standards.html , https://www.rosenlaw.com/pdf-files/DefiningOpenStandards.pdf

204 - https://www.php-fig.org/psr/ و https://github.com/php-fig/fig-standards/tree/master/accepted

205 - PSR-1, PSR-2, PSR-12

206 - https://www.javatpoint.com/software-engineering-structured-programming

207 - https://en.wikipedia.org/wiki/The_Power_of_10:_Rules_for_Developing_Safety-Critical_Code

a. Code review[۲۰۸]
b. Clean Code
c. Code Formater
d. Linter
e. Code coverage (Tests)[۲۰۹]
f. Coding best practices[۲۱۰]
g. Coding conventions[۲۱۱]
   - Camel case
   - Pascal case
   - Snake case
   - Kebab case
   - Middot case
h. Use Plural nouns for resources
i. Anti-Patterns[۲۱۲]

۱۱) نحوه ارتباط با سایر نرم افزارها و مدیریت آن
۱۲) Vendor Packages/Plugins/Library
۱۳) **Interaction and communication**
   a. Shared memory communication **(Java, C#)**
   b. Message passing communication **(MPI, Go, Scala, Erlang/Elixir, Dart, occam)**
۱۴) Version Control
۱۵) Unit tests code
۱۶) Patterns
۱۷) Micro Services
۱۸) MVP[۲۱۳] کمینه محصول پذیرفتنی
۱۹) Proof of Concept[۲۱۴] PoC
۲۰) Prototype[۲۱۵]
۲۱) UX/UI
۲۲) SEF[۲۱۶]/SEO
   a. Server-Side Render

---

208 - https://en.wikipedia.org/wiki/Code_review
209 - **"Leave Things BETTER/CLEANER than you found them."** ~ Robert Baden Powell(The Boy Scout Rule ~Robert C. Martin (Uncle Bob))
210 - https://en.wikipedia.org/wiki/Coding_best_practices
211 - https://en.wikipedia.org/wiki/Coding_conventions , https://en.wikipedia.org/wiki/Letter_case
212 - https://en.wikipedia.org/wiki/Anti-pattern
213 - https://en.wikipedia.org/wiki/Minimum_viable_product
214 - https://en.wikipedia.org/wiki/Proof_of_concept , https://www.techtic.com/blog/proof-of-concept-in-software-development/ https://www.geeksforgeeks.org/software-engineering/what-is-proof-of-concept-poc-in-software-development/ https://www.atlassian.com/work-management/project-management/proof-of-concept
215 - How a prototype differs from a PoC and MVP
   1) **Prototype:** Focuses on user experience and design. It visually demonstrates the product's flow, look, and feel to get feedback on the user interface and interaction design.
   2) **PoC (Proof of Concept):** Focuses on technical feasibility. It's a stripped-down, functional experiment built to prove a specific concept can be implemented. It is not for user testing or demonstrating user experience.
   3) **MVP (Minimum Viable Product):** Focuses on market fit. It's the most basic version of the product with just enough features to be released to early customers for feedback and to validate market demand.
216 - Search Engine Friendly & Search Engine Optimization

b. Client-Side Render

۲۳) MVC

۲۴) Template Engine

۲۵) License

۲۶) محیطهای استاندارد

a. Development

b. Pre-Production/Staging/Testing

c. Production/Live

۲۷) تکنولوژی ها (رجوع به پیوست ۴)

۲۸) When to Compile

```
Change in Application, Business Logic, Refactor    need reCompile
Change in Business Rules in Rule     Engine        no need recompile
Change in Business Flows in Workfolw Engine        no need recompile
Change in Application Config                       no need reCompile
Change in Site        Config                       no need reCompile
Change in License     Config                       no need reCompile
Change in Templates   Config                       no need reCompile
Change in Contents/HTML/CSS/JS                     no need reCompile
Change in /public/assets/*                         no need reCompile
Change in .env(USER/Password, …)                   no need reCompile
Change in Plugins (Add/Remove)                     no need reCompile
Change in Static Pages                             no need reCompile
Change in Tests                                    no need reCompile
Change in Services                                 no need reCompile
Change in Calculation Formula                      no need reCompile
```

API Optimization

- CAP Theorem
- Use **Mapper** to map **Data Transfer Object DTO**[۲۱۷] and **Domain Model**
- Single Tenant or Multi Tenant[۲۱۸] API **(SaaS Architecture Maturity Model)**
- Push (event driven) vs Pull (request/response driven) Architecture [۲۱۹]
- Connection Pool[220]

[217] - https://en.wikipedia.org/wiki/Data_transfer_object , using DTOs for creating contracts between the backend API and the API client. DTO Reducing overhead in remote calls and help Reduction of coupling Service Layer API. DTOs can encapsulate validation logic, ensuring that data entering a particular layer or component meets certain criteria before being processed.

[218] - https://en.wikipedia.org/wiki/Multitenancy , https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/considerations/tenancy-models , https://learn.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns?view=azuresql , https://learn.microsoft.com/en-us/azure/azure-sql/database/elastic-pool-overview?view=azuresql , https://github.com/JonPSmith/AuthPermissions.AspNetCore/wiki/Multi-tenant-explained

[219] - Pull vs Push APIs

- In a pull API, the client requests the information. In a push API, the server sends the information as it becomes available.
- Pull architecture is request driven: the client sends the request, and the server responds accordingly. Push architecture is event driven: the server pushes data to clients as updates become available.
- In a push API, the server needs to store client details to reach clients directly. With pull, the server doesn't need to store these details, because they're encoded in the request.
- Push APIs are significantly faster than pull APIs. In a pull API, the server must receive and verify the request, then process information to form a response that's sent to the client. In a push API, the server immediately processes information and sends it to clients as soon as it's available.

[220] - https://en.wikipedia.org/wiki/C10k_problem

- REST API Best Practices[۲۲۱]
- Avoid Stateful endpoing
- Avoid ORM Lazy Loading
- Avoid Request Proxy by accident
- Avoid Network call in a loop too many
- Avoid N+1 Query Problem[۲۲۲]
- Resiliency
- HATEOAS[۲۲۳]
- Server-Sent Events (SSE)
- Web Sockets
- Idempotency[۲۲۴] (Ensure reliability through idempotent API)
- Add Pagination, Filtering and Logging
- Use clear naming
- Async Logging
- Caching
- Circuit Breaker and Retries
- Use clear query strings for sorting and filtering API data
- Don't make security an afterthought when designing APIs
- Data Serializers[۲۲۵] (JSON[۲۲۶], gRPC, …)
- Huge Payload Compression
- Keep cross-resource references simple
- HTTP and HTTPS and their most common methods
- Return the correct HTTP status codes

---

[221] - https://restfulapi.net/rest-api-best-practices/
[222] - https://restfulapi.net/rest-api-n-1-problem/
[223] - https://en.wikipedia.org/wiki/HATEOAS , https://restfulapi.net/hateoas/
[224] - https://en.wikipedia.org/wiki/Idempotence , https://restfulapi.net/idempotent-rest-apis/

Why is Idempotency Important?
In unreliable networks or distributed systems, requests may fail or experience timeouts, prompting the client to retry the request. Without idempotency, these retries could lead to:

- **Duplicate Records:** Multiple orders or data entries could be created unintentionally.
- **Inconsistent State:** The system might store redundant or conflicting information.

When to Use Idempotency

- **Order Creation**: If an online store allows customers to place orders, retrying an order request could result in duplicate orders. Using an idempotency key prevents this.
- **Financial Transactions and Payment Processing:** Payments and refunds should always be idempotent to avoid duplicate charges. Retrying a payment request without idempotency could lead to multiple charges for the same transaction.
- **Resource Creation:** When creating resources (like orders or accounts), using idempotency prevents duplicates. Creating resources like products, users, or accounts should be idempotent to avoid duplicates.
- **Critical Updates:** For actions that could impact user data or system state, idempotency makes sure repeated requests don't cause unexpected behavior.
- **Booking Systems:** Hotel or airline bookings need idempotency to prevent users from booking the same reservation multiple times due to retries.

Using an idempotency key: You generate a unique value. We recommend using **UUID4/UUID7** to guarantee uniqueness across multiple processes or servers.

[225] - https://quix.io/glossary/data-serialization , https://serde.rs/

Use cases for data serialization:

- **Microservice architecture** - Enables smooth communication between independent services, fostering a decoupled architecture.
- **Data integration** - Facilitates data transformation and conversion during ETL processes, ensuring consistency across diverse sources.
- **Message queues** - Enables data exchange in real-time systems through message queues and event streams, preventing bottlenecks.
- **Big Data processing** - Specialized formats like Avro and Parquet optimize storage and processing of large datasets in distributed computing frameworks.
- **Manufacturing Data Exchange, Sensor Data Processing, Simulation Data Management**

[226] - https://jsonapi.org/

- Avoid deeply nested URLs (Prefer a flatter structure)
- Use a structured error format
  - Content-Type
  - Type
  - Title
  - Status
  - Instance
  - TraceId
  - RequestId
  - Problem Details
- CDN
- Consistent hashing[۲۲۷]
- Consensus algorithm[۲۲۸]
- Security
  - Plan for Rate limiting based on IP, user, action groups, etc
  - Design Allow list rules based on IP, user, action groups, …
  - HTTPS (SSL/TLS)[۲۲۹]
  - Authentication/Authorization
    - Cookies
    - Session Based Authentication[۲۳۰]
    - Token Based Authentication[۲۳۱]
      - JWT-Based Authentication[۲۳۲] (JSON Web Token)
      - Bearer Token
      - OAuth2[۲۳۳]
      - OpenID Connect (OIDC)[۲۳۴]
      - API Keys[۲۳۵]
      - SAML2 (Security Assertion Markup Language)[۲۳۶]
  - Use WebAuthn
  - Check **OWASP** API Security Risks
  - Error Handling
  - API Versioning
  - Use API Gateway
  - Input Validation
  - Use Leveld API Keys

راه کارهای جلوگیری از کاهش کارائی Performance سیستم[۲۳۷]

- **Software Architecture Design Patterns**[۲۳۸]

---

227 - https://en.wikipedia.org/wiki/Consistent_hashing

228 - https://en.wikipedia.org/wiki/Byzantine_fault , https://en.wikipedia.org/wiki/Consensus_(computer_science)

229 - https://en.wikipedia.org/wiki/Transport_Layer_Security , https://www.geeksforgeeks.org/computer-networks/difference-between-secure-socket-layer-ssl-and-transport-layer-security-tls/

230 - Server is responsible for creating and storing the session data.

231 - https://swagger.io/docs/specification/v3_0/authentication/

232 - https://en.wikipedia.org/wiki/JSON_Web_Token , https://www.jwt.io/ , Stateless authentication, Good for **API-first** applications, Consider refresh token rotation, Requires careful handling to avoid XSS vulnerabilities. Server doesn't store any session state. All the necessary data is contained within the token itself, wich is stored on the client. This makes JWTs stateless. In a distributed system where your application runs on multiple servers, all those servers need acess to the same session data. This is typically achieved by using a centralized session store that all servers can access, like Redis or a distributed SQL database.

233 - https://en.wikipedia.org/wiki/OAuth , https://oauth.net/2/ , https://en.wikipedia.org/wiki/List_of_OAuth_providers

234 - https://openid.net/ , https://en.wikipedia.org/wiki/OpenID

235 - https://en.wikipedia.org/wiki/API_key

236 - https://en.wikipedia.org/wiki/Security_Assertion_Markup_Language , https://en.wikipedia.org/wiki/SAML_2.0

237 - Every system tends towards Complexity, Slowness and Difficulty.

238 - https://www.redhat.com/en/blog/14-software-architecture-patterns , https://www.simform.com/blog/software-architecture-patterns/

- Profiling[۲۳۹]
- **Scalability**
  ۱) Capacity Estimation
  ۲) Horizontal Elasticiy[۲۴۰] Replica
  ۳) Clustering
- **Performance**
  ۱) Big O notation[۲۴۱]
  ۲) Minimize blocking calls (Asynchronous Architecture)
  ۳) Divide-and-conquer algorithm[۲۴۲]
  ۴) MapReduce Parallel programming model[۲۴۳]
  ۵) Dynamic programming[۲۴۴]
    - Greedy algorithm[۲۴۵]
  ۶) Optimize disk I/O management
  ۷) Cache at all tiers
    - Sidecar Chaching
    - Chache Chaining
    - Time to Live (TTL) Chaching
  ۸) Reduce round trips
  ۹) Effective use of partitioning at the data tier
  ۱۰) Concurrent[۲۴۶]/Multi-threading/Parallel Programming[۲۴۷]/Async Process
  ۱۱) Pagination
  ۱۲) JIT[۲۴۸]
  ۱۳) انجام Best Practice ها و بهینه سازی کد
  ۱۴) بهینه سازی لایه کلاینت
    - Client-side Load Balancing
    - State Patterns
    - Composite UI Pattern
  ۱۵) بهینه سازی لایه API Gateway
    - Backend for Frontend (BFF)
    - Circuit Breaker
    - Retry Pattern
    - Request Collapsing
  ۱۶) بهینه سازی لایه Load Balancer
    - Geographical Distribution
    - Health Checks
    - Affinity Based Routing
    - Least Connections
  ۱۷) بهینه سازی لایه Application Server
    - SAGA Pattern
    - CQRS

239 - https://en.wikipedia.org/wiki/Profiling_(computer_programming) , https://en.wikipedia.org/wiki/Performance_engineering
240 - **"Elastic"** means that the hardware capacity of your application will automatically grow or shrink depending on the usage your app has.
241 - https://en.wikipedia.org/wiki/Big_O_notation , https://en.oi-wiki.org/basic/complexity/
242 - https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm
243 - https://en.wikipedia.org/wiki/MapReduce
244 - https://en.wikipedia.org/wiki/Dynamic_programming , https://www.chessprogramming.org/Dynamic_Programming http://taggedwiki.zubiaga.org/new_content/cb98ed99aa0935e8e06ecaa01e863833
245 - https://en.wikipedia.org/wiki/Greedy_algorithm
246 - https://en.wikipedia.org/wiki/Concurrency_(computer_science)
247 - https://en.wikipedia.org/wiki/Parallel_programming_model
248 - https://en.wikipedia.org/wiki/Just-in-time_compilation , https://en.wikipedia.org/wiki/Ahead-of-time_compilation

- Proxy Pattern
- Chain of Responsibility

۱۸) بهینه سازی لایه وب سرور و پراکسی سرور

- Page Cache Pattern
- Compression Pattern
- Lazy Loading
- Content Negotiation Pattern

۱۹) بهینه سازی لایه CDN

- Prefetching
- Parallel Requesting
- Edge Computing
- Domain Sharding
- Adaptive Image Delivery

۲۰) بهینه سازی لایه دیتا بیس

- Vertical Scaling
- Sharding[۲۴۹]
- Replication
- Read-Replica Pattern
- Query Object Pattern
- Caching
- Indexing
- Materialized Views
- Denormalization

۲۱) بهینه سازی مفسر/کامپایلر

۲۲) بهینه سازی فریم ورک

۲۳) بهینه سازی معماری و طراحی

۲۴) بهینه سازی API

۲۵) بهینه سازی میکرو سرویسها و Service Discovery

۲۶) مدیریت و کنترل میزان لاگهای تولید شده توسط سیستم

۲۷) بهینه سازی Message Queue & Brokers

۲۸) بهینه سازی Workflow Engine

۲۹) بهینه سازی Rule Engine

کنترل کیفی و نظارت (اندازه گیری/ارزشیابی متریک ها)

۱) انطباق با استانداردهای کنترل کیفیت[۲۵۰]

a. کنسرسیوم CISQ[۲۵۱]
b. ISO/IEC 9126[۲۵۲]
c. ISO/IEC 25002:2024
d. ISO/IEC 25019:2023 → ISO/IEC 25010:2023 → ISO/IEC 9126
e. ISO/IEC JTC 1/SC 7[۲۵۳]
f. FURPS[۲۵۴]
g. ISO 9001

249 - https://learn.microsoft.com/en-us/azure/architecture/patterns/sharding
250 - https://en.wikipedia.org/wiki/Software_quality
251 - https://en.wikipedia.org/wiki/CISQ
252 - https://en.wikipedia.org/wiki/ISO/IEC_9126
253 - https://en.wikipedia.org/wiki/ISO/IEC_JTC_1/SC_7 , https://www.iso.org/committee/45086.html/
254 - https://en.wikipedia.org/wiki/FURPS

i. Functionality
ii. Reliability
iii. Usability
iv. Efficiency
v. Maintainability
vi. Portability
h. SEICMM
i. PCMM
j. Six Sigma
i. DMAIC
ii. DMADV

۲) قابلیت ردیابی Trace ability
۳) درستی Correctness
۴) اعتبار Validity
۵) کفایت Sufficiency
۶) سازگاری Consistency
۷) یکنواختی Uniformity
۸) نگهداشت پذیری Maintainability
۹) مقیاس پذیری Scalability
۱۰) مدلها[۲۵۵]

a. Capability Maturity Model (CMM)(5 Levels)
b. McCall's Quality Model
c. Boehm's Software Quality Model
d. COCOMO (Cost Constructive Model) Model
i. Basic Model
ii. Intermediate Model
iii. Detailed Model

امنیت[۲۵۶]

تست امنیت نرم افزار دارای ۴ گام اساسی است :

- Static application security testing (SAST) or static code analysis
- Dynamic application security testing (DAST)
- Combining both DAST and SAST approaches is the domain of Interactive Application Security Testing (IAS)
- Software dependency and libraries that have known vulnerabilities

[255] - https://learn.microsoft.com/en-us/azure/well-architected/what-is-well-architected-framework#adopt-a-maturity-model

[256] - https://en.wikipedia.org/wiki/Application_security , https://en.wikipedia.org/wiki/Zero_trust_architecture
https://en.wikipedia.org/wiki/Open_Source_Security_Foundation
https://en.wikipedia.org/wiki/NIST_Cybersecurity_Framework
https://en.wikipedia.org/wiki/Information_security
https://en.wikipedia.org/wiki/Information_security_standards
https://en.wikipedia.org/wiki/Computer_security
https://en.wikipedia.org/wiki/OWASP
OWASP Top Ten: 2025 https://github.com/OWASP/www-project-top-ten/blob/master/index.md

برخی از اجزای آن نیز به شرح ذیل می باشد:[۲۵۷]

۱) Authentication **(Who are you?)**
۲) Authorization **(What are you allowed to do?)**
۳) Encryption/Hash Algorithms
۴) Penetration Test
۵) SQL Injection
۶) Hardening
   a. Keep update
   b. Hard password to guess
   c. Backup
   d. ....
۷) Over Flow
۸) PHP Sodium Extension + Hasher/Encryption Algorithms

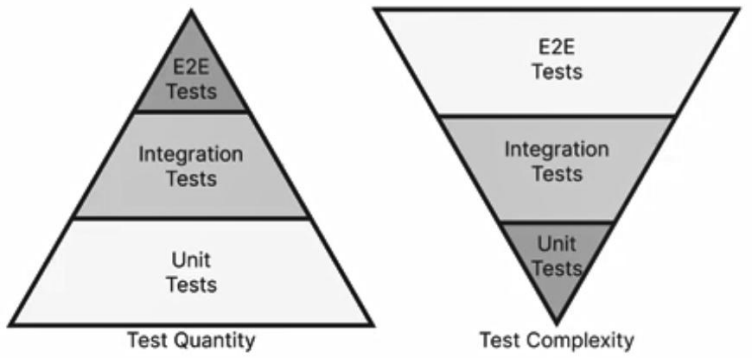
چرخه تست نرم افزار[۲۵۸]

تست معمولا به مواردی همچون Testing, Debugging, Exception Handling اطلاق میشود.

تست فقط وجود خطا را نشان می‌دهد و نه عدم وجود آن را. پیدا نشدن خطا در تست به معنای بدون خطا بودن برنامه نیست.

تست غالبا به ۲ سطح تقسیم میشود:

- **Functional testing**
  - Unit test
  - Component test
  - Integration test
  - System test
  - Acceptance test (end to end)
- **Nonfunctional testing**
  - Security test
  - Performance test
  - Regression test
  - Usability test
  - Compatibility test



جزئیات بیشتر درذیل لیست شده است:

۱) تست سند گزارشات بیزینسی

[257] - https://sec.ito.gov.ir/ , https://paydarymelli.ir/ برای اطلاعات بیشتر رجوع کنید به

[258] - https://en.wikipedia.org/wiki/Software_testing

۲) تست آماره های اپلیکیشن/سایت

۳) سطوح تست :

a. **سند تست واحد (Unit testing[۲۵۹]) :** تست واحد یا micro level پایین‌ترین سطح تست است. هر کد تست واحد، یک قطعه کد یا یک تابع (متد) خاص را تست می‌کند. این تست نیاز به دانش در مورد طراحی و نحوه کارکرد داخلی تابع یا قطعه کد دارد[۲۶۰] و توسط برنامه‌نویس **(و نه تست‌کننده)** انجام می‌شود. **در این تست صرفاً ورودی های مشخصی به توابع داده شده و صحت خروجی های مورد انتظار تست می گردد.** هنگامی که توسعه توسط روش TDD[۲۶۱] انجام میگیرد می توان از این نوع تست سود برد.

b. **سند تست یکپارچگی افزایشی :** تست یکپارچه‌سازی افزایشی با افزوده شدن قابلیت جدید به نرم‌افزار، مجدداً نرم‌افزار تست می‌شود. هدف این تست، بررسی درستی نرم‌افزار پس از افزوده شدن امکان جدید است.

c. **سند تست یکپارچگی (Integration Testing[۲۶۲]) :** همانطور که از اسم آن نیز پیداست از این روش برای تست صحت کارکرد درست بین اجزای/بخشهای مختلف سیستم استفاده می شود. این تست بر روی بررسی ارتباط داده‌ها(Data Communication) در میان ماژول‌های مختلف متمرکز است. تست یکپارچه‌سازی برای تایید ماژول‌های نرم‌افزاری برای کار در یک پیکر واحد ضروریست[۲۶۳]. برخلاف تست واحد که اجزا به صورت مجزا و ایزوله تست میشوند در اینجا هماهنگی بین کل اجزا سیستم شرط اساسی است. بدلیل پیچیدگی پروسه، انجام تست های این بخش کندتر از تست واحد صورت می گیرد. تعداد تست های این بخش کمتر از تعداد تستهای واحد می باشد. در برخی موارد انجام تست های این بخش توسط همان ابزار تست واحد نیز میتواند انجام گیرد.

d. **سند تست سیستم (System Testing[۲۶۴]) :** به منظور بررسی عملکرد نرم‌افزار بر روی پلتفرم‌های مختلف انجام می‌شود.

1) Page testing
2) Cross-page testing
3) Logic testing
4) Linting (isn't about finding errors, but potential errors)
5) Link checking (making sure there are no broken links on your site)

---

259 - Test Methods of a class. Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

260 - Unit test included but not limited to below list:
1) **Overflows and underflows:** Make sure that your code doesn't allow numbers to become larger than the largest valid value or smaller than the smallest valid value. Either situation will cause an error.
2) **Valid return values:** Ensure that each function returns values that are valid for the caller. In some cases, the return value is calculated. Your tests should ensure that any calculated return values are always valid.
3) **Boundary conditions:** Always test that your code handles data that meets or exceeds expected limits.
4) **Iteration limits:** Test each looping structure to ensure that it doesn't iterate more times than you intend and burn up all your gas.
5) **Input and output data formats:** Test your code to make sure that it handles data provided or returned in unexpected formats.
6) **Input and output data validation:** Ensure that your code either sanitizes or rejects invalid characters or sequences of characters.

261 - Test driven development (TDD)
Behavior driven development (BDD)

262 - Checks if Class A works with Class B and Ensure different parts of system work together.

263 - Integration test included but not limited to below list:
1) Wrong function name
2) Wrong number or format of input parameters
3) Out of range or bad input data
4) Input data containing boundary values
5) Wrong expected output parameters
6) Attempt to call a function that isn't visible
7) *Smart contract* function properly completed with return codes
8) Set a timeout for a function call that is too short
9) Reverse a transaction

264 - System testing ensures the whole system works as user expected before sending it to acceptance testing.

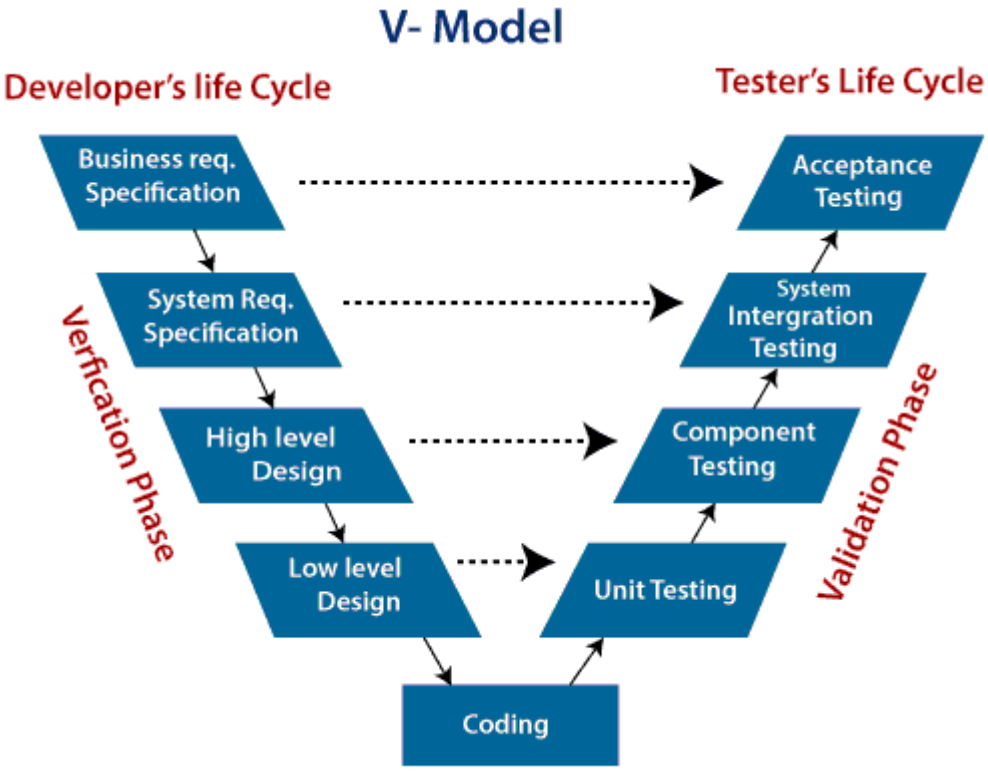
e. **سند تست پذیرش (Acceptance Testing[۲۶۵])** : هدف از این آزمون اطمینان از این نکته است که سیستم در شرایط عملیاتی معمولی و با اطلاعات واقعی قادر به برآورده کردن نیازهای کاربران می باشد.

۶) ابزارهای تست

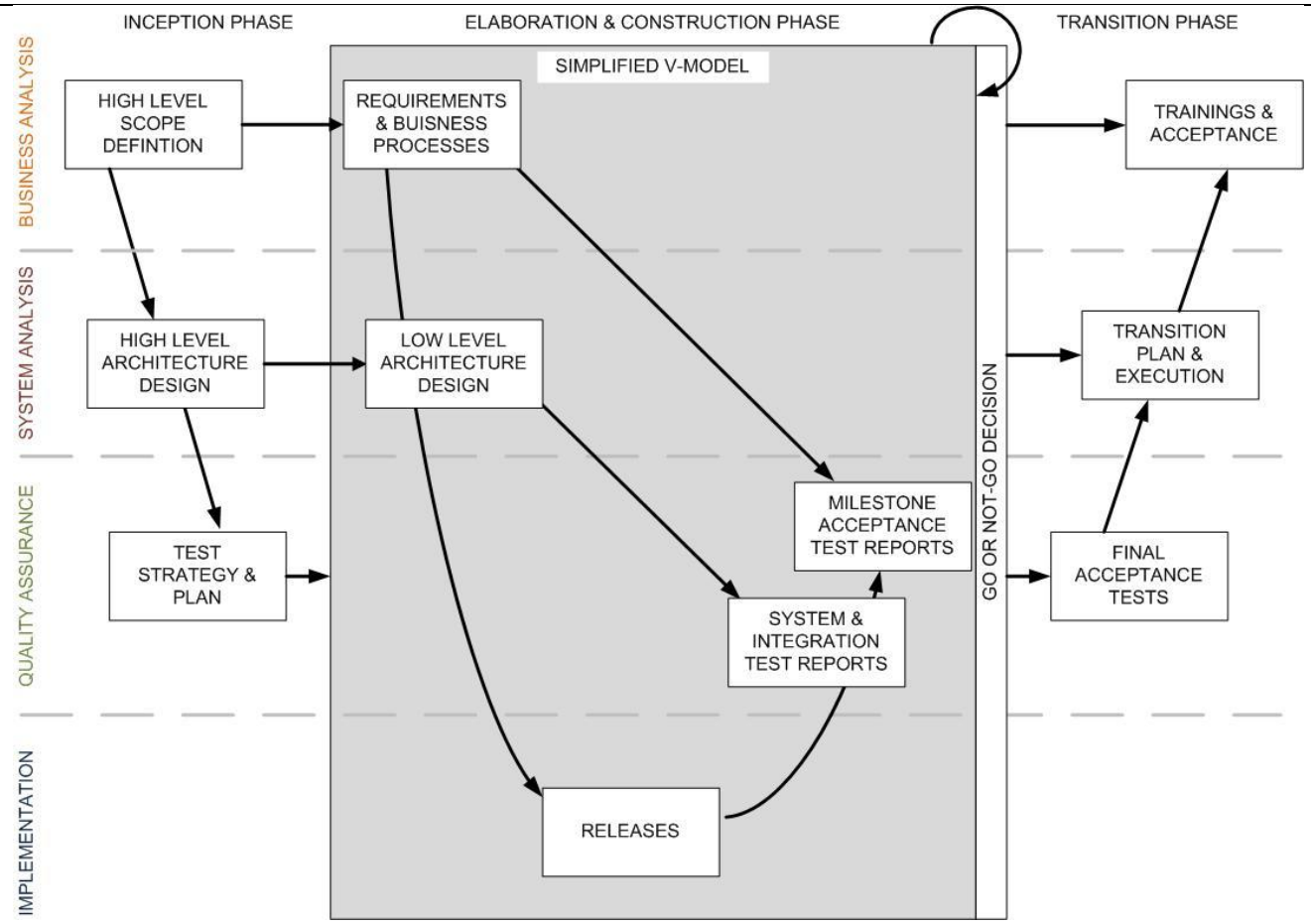
۷) آزمون استقلال سیستم از پلتفرم (Linux, Windows, Mac,…)

۸) آزمون استقلال سیستم از مرورگر (Chrome, Firefox, Edge, IE, Safari,…)

۹) آزمون اندازه‌های مختلف صفحه نمایش.



[265] - When tests above are for developers at development stage. Acceptance tests are actually done by the users of the software. Users do not care about the internal details of the software. They only care how the software works.



۱۰) آزمون کارکردی (مشخصات پیش بینی شده در اهداف ورودی و خروجی کارکردی)

۱۱) آزمون عملکرد (کارکردی با هزینه(مصرف زمان و منابع) قابل قبول. آستانه پذیرش کارایی سیستم در هر کارکرد باید با توافق کاربر نهایی تعیین گردد.)

۱۲) آزمون همسازی داده ها Data integrity (در صورت کار دائم سیستم، هیچ یک از Constraintها نقض نشود.)

۱۳) آزمون چرخه کسب و کار Business cycle

۱۴) آزمون واسط کاربر GUI (E2E Test, Browser/ Functional testing)

۱۵) آزمون امنیت

۱۶) تست نفوذ Penetration test

۱۷) آزمون تحمل خرابی Fault-tolerance (Planned/Unplanned crash recovery)

۱۸) آزمون پیکربندی

۱۹) آزمون بازگشتی Regression (پس از هربار ارائه یک نسخه جدید از سیستم)

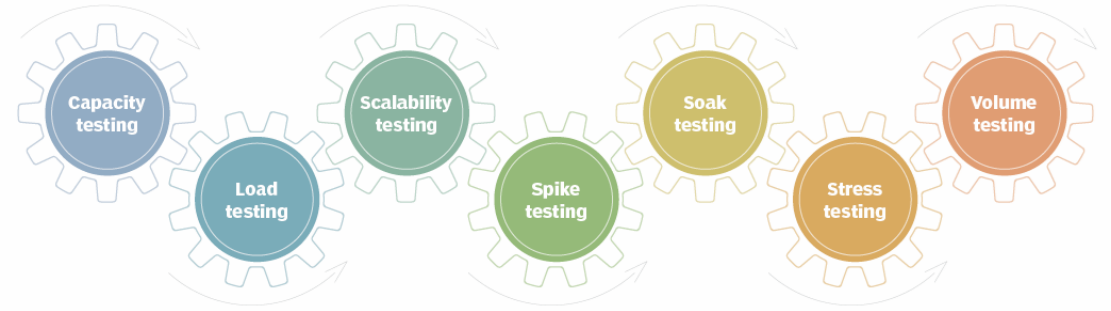
a. تصحیحات انجام شده، منجر به رفع اشکالات قبلی یا بهبود کارایی سیستم شده است.

b. تصحیحات انجام شده، منجر به بروز اشکالات جدید در دامنه پوشش آزمونهای قبلی نشده است.

۲۰) آزمون تحمل بار و تنش

a. Load Test
b. Stress Test
c. Spike Test
d. Soak Test
e. Endurance Test
f. Volume Test

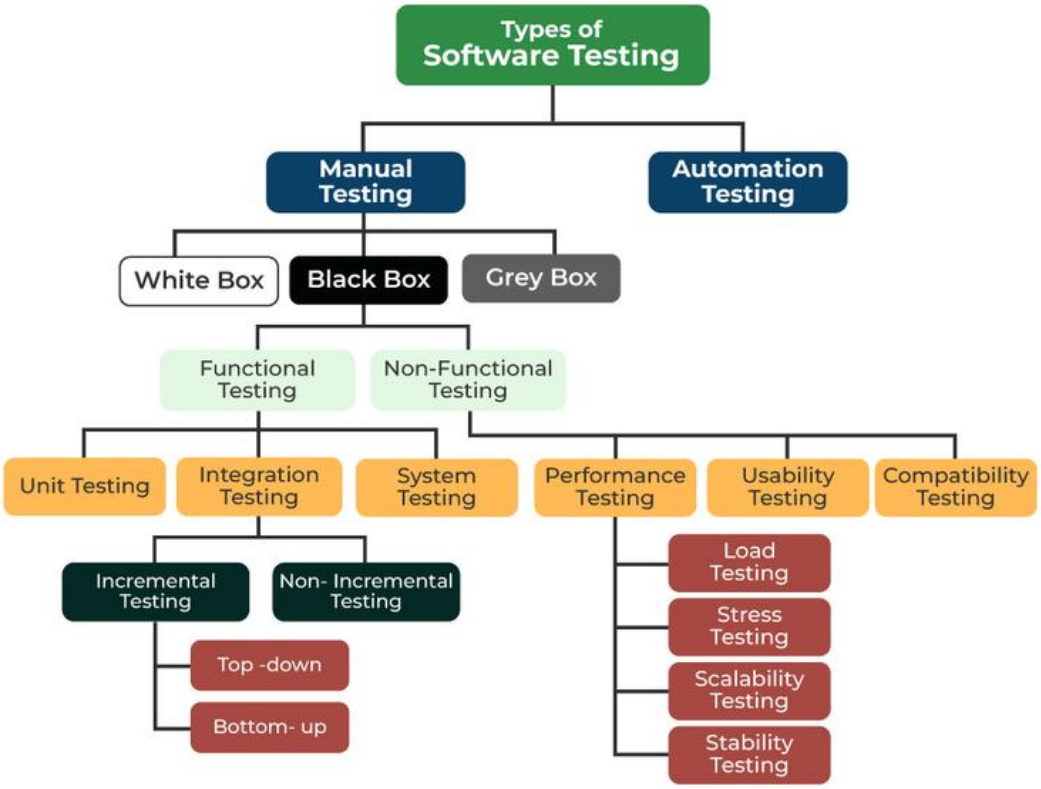
g. Scalability Test



۲۱) Alpha/Beta Testing

۲۲) تست API طراحی شده برای نرم افزار.

۲۳) Pre-flight check : اطمینان از اینکه سیستم در محیط عملیاتی نیز همانند محیط تست، بدون مشکل اجرا خواهد شد. تست این بخش معمولا شبیه Smoke testing می باشد.



فاز ۴) استقرار محصول/ سرویس و نظارت بر عملکرد کیفی(NOC) و امنیت(SOC) آن

۱) سند نحوه انتقال و واگذاری نرم‌افزار
۲) سند نحوه نگهداری و به روز آوری (Update/Upgrade/Patch/BugFix/Refactoring) سیستم
۳) سند تفاوتهای پیکربندی نسخه Development با نسخه Production
۴) سند تفاوتهای پیکربندی Host, Localhost
۵) ابزارهای Upload/Download/Backup
۶) سند پیکربندی DNS
۷) سند پیکربندی eMail
۸) سند نحوه مانیتورینگ، NOC/SOC
۹) سند Logs و Log circulation

ابزارها و نحوه پیکربندی آنها

۱) کانتینرها (Docker,...)
۲) پایگاه داده MySQL, MariaDb, ...
۳) مفسر PHP, NodeJS, ...
۴) وب سرور Apache, Nginx, ...
۵) پلاتفرم
۶) فریم ورکها
۷) API Gateways
   a. Kong
   b. APISIX
   c. Tykio
   d. APIgee
۸) Message Brokers
   a. Kafka
   b. Pulsar
   c. NATS
   d. RabbitMQ
   e. Redis
   f. ZeroMQ, Aeron[۲۶۶]
۹) Template engines
۱۰) ابزار مدیریت وابستگی ها (Composer ,NPM ,...)
   a. Package/Plugins های مورد نیاز
   b. 3rd Party Library
۱۱) ابزارها و سیاستهای بکاپ گیری

اسناد کیفیت ارائه سرویس SLA (Service-level agreement)

۱) Planned down time
۲) Unplanned down time
۳) Crash/Recovery Policy

[266] - https://zeromq.org/ , https://aeron.io/

۴) Restore & Resolution time
۵) Retention time
۶) Ticketing
۷) سند پروتکل ارتباط با مشتریان، نحوه ارجاع مشکلات به واحد فنی(L2,L3) و پروسه رفع آن

تحویل دادنی ها

۱) نسخه نهائی سورس کد نرم افزار(به صورت Optimum[۲۶۷])
   a. **تحویل آزمایشی[۲۶۸] (به صورت هفتگی)** (محیط آزمایشی،تست کارکردی و عملکردی با داده های تستی و واقعی)
   b. **تحویل موقت(فاز انتهایی پروژه)** (محیط عملیاتی، عملیاتی شدن سیستم و تست پایداری سیستم و بررسی باگهای احتمالی(سه الی شش ماه))
   c. **تحویل دائم(خاتمه پروژه)** (گذراندن کلیه تستهای آزمایشی و عملیاتی و پایدار شدن سیستم)
۲) پایگاه اطلاعاتی سیستم(اسکریپت ایجاد، Dump)
۳) راهنمای نصب و استقرار
۴) راهنمای کاربران
۵) آموزش کاربران و FAQ
۶) راهنمای عملیاتی سیستم
۷) طرح آزمون پذیرش
۸) کلیه اسناد توسعه نرم افزار[۲۶۹]
   a. اسناد الزامات کسب و کار
   b. اسناد الزامات عملکردی
   c. اسناد الزامات بازار
   d. اسناد الزامات استانداردها(محیطی، داخلی)
   e. اسناد الزامات محصول
   f. اسناد الزامات رابط کاربری
   g. اسناد الزامات فنی
   h. اسناد مشخصات الزامات نرم افزار
   i. اسناد الزامات کیفیت
   j. اسناد الزامات تست نرم افزار
   k. اسناد الزامات امنیتی نرم افزار
      - Static Analysis
      - Dynamic Analysis
      - Interactive Application Security Testing
      - Software Composition Analysis (SCA)
   l. اسناد الزامات مشتری

فاز ۵) نگهداری و پشتیبانی

لیست زیر بخش از مواردی است که نیاز به نگهداری و پشتیبانی را لازم می سازد:

۱) اصلاح مشکلات گزارش شده

---

[۲۶۷] - نسخه ای کامل است که نتوان چیزی از آن کم کرد.

[۲۶۸] - Demo به صورت هفتگی

[۲۶۹] - اگر نرم افزار جهت فروش/صادرات باشد نیازی نیست، ولی چنانچه "خریدار/مشتری/کاربرنهائی/کارفرما/بهره بردار" خود بایستی ادامه توسعه و نگهداری را عهده دار باشد، ارائه کلیه مستندات توسعه نرم افزار توسط "پیمانکار/مجری"، ضروری خواهد بود.

۲) تغییر در نیازمندیهای کاربران
۳) تغییر در نیازمندیهای سخت افزاری و نرم افزاری
۴) بهبود کارائی
۵) اصلاح عناصر برنامه
۶) کاهش عوارض جانبی ناخواسته
۷) **بدهی فنی انباشته** در طول فاز توسعه نرم افزار

انواع سیاستهای نگهداری :

۱) Corrective Maintenance
۲) Adaptive Maintenance
۳) Preventive Maintenance
۴) Perfective Maintenance

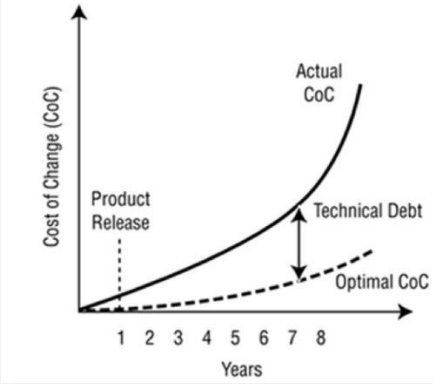
بدهی فنی[۲۷۰] Technical Debt

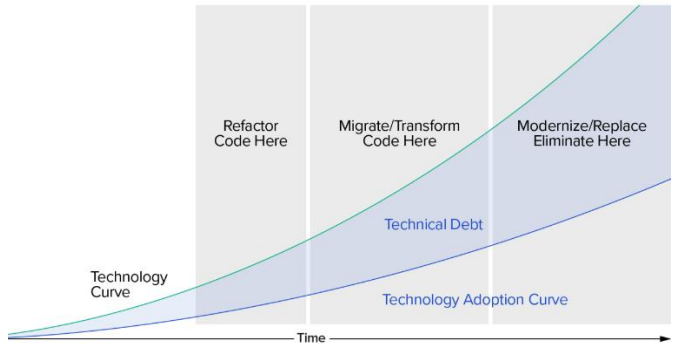
بدهی فنی، مفهومی در توسعه نرم‌افزار است که نشان دهنده هزینه ضمنی کار اضافی ناشی از انتخاب یک راه حل آسان (محدود) در زمان حال به جای استفاده از یک رویکرد بهتر است که طولانی‌تر خواهد بود. دلایل عمده بدهی فنی (بدهی کد، بدهی طراحی) شامل موارد زیر است:

۱) تعریف ناکافی از چشم‌انداز کار، جایی که هنوز الزامات در طول توسعه تعریف می‌شود، توسعه قبل از انجام هرگونه طراحی شروع می‌شود. این کار برای صرفه جویی در وقت انجام می‌شود اما اغلب در آینده نیاز به دوباره کاری می‌شود.
۲) فشارهای بیزینسی، جایی که بیزینس در نظر دارد زودتر از اتمام همه تغییرات لازم، محصول را منتشر کند، بدهی فنی‌ای را تشکیل می‌دهد که شامل تغییرات ناقص است.
۳) عدم پردازش یا تفاهم، در شرایطی که کسب‌وکارها مفهوم بدهی فنی را نمی‌دانند و بدون در نظر گرفتن پیامدهای آن تصمیم‌گیری می‌کنند.
۴) کامپوننت‌های به هم تنیده، در صورتی که سیستم‌ها ماژولار نباشند، نرم‌افزار به اندازه کافی انعطاف‌پذیر نیست تا بتواند با نیازهای تجاری سازگار شود.
۵) عدم وجود تست کیس‌ها، که برای تشخیص سریع مشکلات استفاده شود.
۶) عدم وجود مستندات، در جایی که کد بدون مستندات لازم پشتیبانی، ایجاد شود. کار برای ایجاد هرگونه مستندات پشتیبانی در آینده، بیانگر بدهی است که باید پرداخت شود.
۷) عدم همکاری، در جایی که دانش در مورد سازمان به اشتراک گذاشته نمی‌شود و کارایی کسب‌وکار پایین است، یا توسعه دهندگان کم تجربه به درستی راهنمایی و مربیگری نمی‌شوند.
۸) توسعه موازی در دو یا چند برنچ/شاخه به دلیل کار مورد نیاز برای ادغام تغییرات در یک منبع واحد، بدهی فنی را ایجاد می‌کند. هرچه تغییرات بیشتر در انزوا انجام شود، بدهی بیشتری جمع می‌شود.
۹) تأخیر در ریفکتورینگ، از آنجا که الزامات یک پروژه در حال تحول است، ممکن است مشخص شود که قسمت‌هایی از کد ناکارآمد یا نگهداری آن دشوار شده‌است و برای پشتیبانی از نیازهای آینده باید مورد بازبینی قرار داده شود. هرچه این موضوع مدت طولانی‌تری به تأخیر انداخته شود و کد بیشتری اضافه شود، بدهی نیز بیشتر می‌شود.
۱۰) عدم مطابقت با استانداردها، جایی که ویژگی‌های استاندارد صنعت، فریم‌ورک‌ها و فناوری‌ها نادیده گرفته می‌شوند. سرانجام زمان ادغام با استانداردها فرا خواهد رسید. هر چه انطباق با استانداردها زودتر انجام شود هزینه کمتری خواهد داشت.
۱۱) عدم دانش، هنگامی که توسعه دهنده به سادگی نمی‌داند چگونه کد تمیزی بنویسد.
۱۲) عدم مالکیت، هنگامی که تلاش‌های برون سپاری بخشی از کارها منجر به این می‌شود که تیم داخلی نیاز به ریفکتورینگ یا بازنویسی کد برون سپاری شده را پیدا می‌کند.

[270]- https://en.wikipedia.org/wiki/Technical_debt

۱۳) رهبری ضعیف فنی، زمانی که تیم از مدیریت فنی ضعیفی برخوردار باشد بدهی‌های فنی افزایش می‌یابد.
۱۴) تغییرات لحظه آخری در نیازمندی‌ها، این امکان وجود دارد که در طول یک پروژه برای اعمال تغییرات تحت فشار قرار بگیریم اما زمان‌بندی و بودجه بندی مناسب برای پیاده‌سازی آن‌ها در نظر گرفته نشود.





اصلاح کد (کاهش بدهی فنی و افزایش اعتماد Confidence)

برای اصلاح مشکلات گزارش شده سورس کد میتوان به روشهای زیر عمل نمود:

- Update/Upgrade/Patch/BugFix
- Refactor (Repaying Technical Debt[271])
- Port
- Reengineering/Migrate/Transform
- Replace/ReWrite[272]
    - Decomposing monoliths into microservices ***(Trunk-Based Development and Branch by Abstraction)***[273]
        i. Decompose by business capability
        ii. Decompose by subdomain
        iii. Decompose by transactions
        iv. Service per team pattern
        v. Strangler fig pattern
        vi. Branch by abstraction pattern

شرح موارد فوق:

۱۲) **Refactor**[274] : این روش برای بهبود ساختار کد، بدون هیچ تغییری در رفتار سیستم (و منطق آن) می باشد به عبارتی دیگر، اصلاح کد بدون هیچ تغییری در Functionality و صرفا بهبود خوانائی و ارتقا سرعت و NonFunctionality آن. زبان برنامه نویسی و ساختار هیچ تغییری نخواهد داشت. این عمل ریسک کمی داشته و غالبا به سادگی انجام میشود.
۱۳) **Port**: در این روش معمولا معماری و طراحی هیچ تغییری نخواهد داشت. زبان برنامه نویسی، فریم ورک و یا تکنولوژی بکار رفته، تغییر و یا ارتقا خواهد یافت. این بهبودها نبایستی هیچ تغییری در رفتار سیستم و منطق آن ایجاد کند. این روش دارای ریسک متوسط است.
۱۴) **ReEngineering/Migrate/Transform**: این روش جهت ارتقا ظرفیت، ماجولاریتی، انعطاف و مقیاس پذیری سیستم انجام میگیرد. بدیهی است که برای این منظور معماری و طراحی سیستم دستخوش برخی تغییرات خواهد شد. باز مهندسی های اعمال شده، نبایستی هیچ تغییری در رفتار سیستم و منطق آن ایجاد کند.

271 - https://github.com/mistermicheels/learning-notes/blob/master/processes-techniques/Managing-technical-debt.md
272 - https://en.wikipedia.org/wiki/Rewrite_(programming)
273 - https://martinfowler.com/bliki/BranchByAbstraction.html
https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/welcome.html
https://learning-notes.mistermicheels.com/processes-techniques/branch-by-abstraction-application-strangulation/
274 - Refactoring is the process of improve and reorganizing code without affecting its original functionality.
https://en.wikipedia.org/wiki/Code_refactoring

۱۵) **ReWrite/Replace**: این روش برای بازنویسی سیستم استفاده می شود. بدیهی است که کلیه بخشها (معماری، طراحی، فریم ورک، زبان برنامه نویسی، ...) تغییرات اساسی خواهد داشت. بازنویسی های اعمال شده، نبایستی هیچ تغییری در رفتار سیستم و منطق آن ایجاد کند(مگر در نسخ آتی). این رویکرد با ریسک زیاد و چالش های بسیاری همراه است، از جمله :

a. Loosing Functionality
b. Loosing Bug Fixes
c. Late Return on Investment
d. Hidden Additional Costs
e. Second-System Effect

انواع پرداختها

پروسه تحویل گیری پروژه های بزرگ معمولا طولانی و بتدریج می باشد، که پیشنهاد میشود شامل ۱۵ زیر بخش لیست زیر باشد (تا از ریسکها و مشکلات سایر پروژه های مشابه دوری جست):

۱) پیش پرداخت اولیه

۲) پرداخت بر اساس پیشرفت پروژه (Milestone ها و Sprint ها)

۳) پرداخت تحویل سورس کد و مستندات توسعه دهندگان نرم افزار

- مستندات طراحی نیازمندیها Requirements.
- مستندات طراحی تفضیلی و معماری.
- مستندات و سورس کد طراحی دیداری.
- مستندات و سورس کد نمونه اولیه MVP.
- مستندات و سورس کد طراحی و تست امکانات کارکردی اولیه/اجباری .
- مستندات و سورس کد طراحی و تست امکانات کارکردی ثانویه و گزارشات.
- مستندات و سورس کد API و سرویسهای مرتبط.
- مستندات و سورس کد پلاگینها.
- مستندات و سورس کد طراحی و تست امکانات عملکردی(از نظر ... ,Performance) .
- مستندات و سورس کد طراحی تست کلی نرم افزار و تست نفوذ.
  i. Unit test
  ii. Integration test
  iii. System test
  iv. Acceptance test

۴) پرداخت مستندات آموزشی کاربران.

۵) پرداخت تحویل موقت.

۶) پرداخت پس از دوره گارانتی و تحویل دائمی.

۷) پرداخت تحویل مستندات و سورس طراحی نحوه استقرار، نگهداری و **پشتیبانی کسب و کار** و سطح کیفیت سرویس SLA بصورت روزانه/هفتگی/ماهانه/سالانه.

۸) پرداخت نگهداری، توسعه و **پشتیبانی فنی** مورد نیاز آتی.

دلایل شکست پروژه ها

سازمان پروژه معمولا به ۳ لایه استراتژی، تاکتیکی و تکنیکی تقسیم میشود که لایه های بالاتر ماموریت و متریکهای لایه پائین تر را تعریف و بر حسن اجرای آن نظارت میکند. در لایه استراتژیک بایستی به دنبال موارد زیر بود:

- **مشکل اصلی چیست؟ هدف از حل این مشکل چیست؟ قرار است به چه نیازی پاسخ داده شود؟!**
- فاکتورها، عوامل، پارامترها و معیارهای اساسی و بنیادین چیست؟
- شناخته ها و ناشناخته ها چیست؟
- امکانات، منابع، محدودیتها و الزامات چیست؟
- راه حل های ممکن کدام است؟
- کدام راه حل ها، انعطاف پذیر هستند و یا منجر به نتایج مطلوب تر میشوند؟
- چگونه این راه حل ها ارزیابی و اعتبار سنجی می شوند؟
- کدام راه حل بهینه است؟
- روش اندازه گیری نتایج (و یا انحراف) بدست آمده و میزان فاصله تا هدف مورد انتظار چیست؟!

علل استراتژیک/راهبردی شکست پروژه

۱) خوش بینی بیش از حد به ایده اولیه پروژه و فرصت ساز نبودن خروجی نهائی پروژه. اکثر ایده ها واقعا ارزش اجرائی نداشته و ارزش افزوده واقعی خاصی ایجاد نمی کنند. ویژگی های یک ایده خوب که منجر به یک خروجی/محصول خوب میشود معمولا شامل موارد زیر است:
   - یک مسئله/مشکل را حل کند.
   - مشتریان حاضر باشند بخاطر آن هزینه کنند.
   - تا جائی که ممکن است متفاوت باشد.
   - قابل رشد و مقیاس پذیر باشد.
   - تقاضای بالائی برای آن وجود داشته باشد.
   - براحتی قابل کپی کردن نباشد.

۲) نبود تحقیقات کافی

۳) ضعف در سازمان پروژه[۲۷۵] (به پیوست ۶ مراجعه شود) و استفاده از استانداردهای موجود

۴) انتظارات غیر واقع بینانه

۵) تخصیص منابع ناکافی (مالی، انسانی، فنی، ...)

۶) ضعف در هدف گذاری (مشخصات محصول) و بازار هدف(نوع مشتریان)

۷) ضعف مدیریت در تعهد طولانی مدت، فقدان تعهد و ضمانت اجرایی

۸) برنامه ریزی ضعیف پروژه

۹) کمیته راهبری ضعیف

۱۰) مدیریت زمان، هزینه، ریسک

۱۱) فقدان چشم انداز روشن

۱۲) ضعف در تعریف فرآیندهای سازمان

۱۳) عدم ایجاد توازن بین پروژه های سبد (Project Portfolio Balancing[۲۷۶])

۱۴) ضعف در برند سازی و موفقیت محصول/پروژه

۱۵) همیشه در حال برنامه ریزی بودن ، بدون اقدام مناسب و یا اقدامات زیادی بدون برنامه ریزی مناسب

۱۶) عدم تمرکز کافی

[275] - What one programmer can do in one month, two programmers can do in two months.

[۲۷۶] – به معنی توزیع منابع بین پروژه‌های مختلف به شکلی متعادل، بدون تمرکز بیش از حد روی یک پروژه و غفلت از سایر پروژه‌های مهم می باشد. شامل ترکیب مناسبی از پروژه‌های کوتاه‌مدت و بلندمدت، پرریسک و کم‌ریسک، و پروژه‌های مختلف با اهداف استراتژیک متفاوت (مانند درآمدزایی، نوآوری، یا بهبود فرآیندها)

علل تاکتیکی شکست پروژه

مشکلات محیطی

۱) منابع ناکافی/محدود
۲) رقابت شدید (عدم توانائی در رقابت با سایر رقبا و محصولات مشابه)
۳) عدم نیاز بازار

مشکلات داخلی

۱) **انباشت بدهی های فنی سیستم/پروژه**
۲) کم توجهی به نیازمندیهای کیفیت محصول **(NonFunctional)**
   a. عدم مقیاس پذیری
   b. عدم یکپارچگی
   c. عدم قابلیت نگهداری
   d. کم اهمیت پنداشتن سناریو های خرابی Failure
   e. عدم ایجاد توازن و بالانس مناسب (نقاط تعادل) **Tradeoffs and Priorities**
   f. ...
۳) خطاها و سوءگیرهای شناختی: **"به پیوست ۵ مراجعه شود"**
۴) تخمین های مدیریت پروژه[۲۷۷]
۵) عدم تصویر یکسان کارفرما و مشتری در خصوص محصول نهائی
۶) ارتباطات ضعیف
۷) شکاف دانش[۲۷۸] میان تیم های مختلف پروژه
۸) ضعف در تصمیم گیری (Lack of clear ownership) . چه کسی بایستی تصمیم بگیرد که چه کار بایستی انجام شود.
۹) نادیده گرفتن و یا کم اهمیت دادن به طراحی سطح بالا
۱۰) یکراست به سراغ سلوشن رفتن و عدم بالانس/تعادل بین سلوشن های گوناگون
۱۱) عدم آمادگی و ضعف در آماده سازی
۱۲) بینش یا هدف ناکافی برای پروژه و یا عدم دستیابی به اهدف پروژه
۱۳) ایجاد انتظارات و توقعات بیش از حد
۱۴) عدم آشنائی پیمانکار از حوزه کسب و کار کارفرما
۱۵) عدم تعریف درست کارفرما از نیازمندیهای اساسی، استانداردهای مهندسی و معیارهای پذیرش محصول نهائی که توسط پیمانکار بایستی اجرا و پیاده سازی شود.
۱۶) غفلت از تاثیرات جانبی، موارد پنهان و غیر قابل لمس که موفقیت پروژه ها را تحت تاثیر قرار میدهد.
۱۷) ضعف در تحلیل ریسکهای موجود، عدم شناخت کافی نسبت به موانع و عدم جلوگیری از واگرائی ها، عدم ارائه راه حل و پوشش ریسکهای احتمالی :
   a. اشتباه در زمانبندی پروژه
   b. افزایش پیچیدگی
   c. افزایش چالشها
   d. افزایش تقاضای مشتریان و عدم برآورده سازی نیازهای کاربران

---

277 - https://en.wikipedia.org/wiki/Estimation_(project_management)
https://en.wikipedia.org/wiki/Software_development_effort_estimation
https://en.wikipedia.org/wiki/Cost_estimation_in_software_engineering
https://en.wikipedia.org/wiki/Cost_estimation_models

278 - https://en.wikipedia.org/wiki/Knowledge_gap_hypothesis

e. کیفیت پائین محصول نهائی
f. برآورد نادرست هزینه‌های آشکار و غفلت از هزینه های پنهان
   i. هزینه های فنی
      ۱. Module Independence
      ۲. Programming Language
      ۳. Programming Style
      ۴. Program Validation and Testing
      ۵. Documentation
      ۶. Configuration Management Techniques
   ii. هزینه های غیر فنی
      ۱. Application Domain
      ۲. Staff Stability
      ۳. Program Lifetime
      ۴. Dependence on External Environment
      ۵. Hardware Stability

۱۸) ضعف در تعریف فرآیندهای پروژه ها
۱۹) وابستگی به منابع (مالی، انسانی، ...)
۲۰) وابستگی وظایف
۲۱) نیازمندی های گمراه کننده
۲۲) عدم برنامه ریزی منابع
۲۳) ضعف استراتژی و مدیریت بازاریابی محصول
۲۴) فقدان دیدگاه مشتری مداری
۲۵) مدیریت تغییر غیر اثربخش
۲۶) ضعف در نظارت بر صحت و دقت انجام امور و اصلاح مسیر و فرآیندها
۲۷) تغییر اولویت بندی ها
۲۸) فقدان و ضعف در سطح مشاوران
۲۹) **عدم وجود جهتی واحد** : بیش از هر چیز، یک تیم موثر به یک هدف مشترک نیاز دارد. پروژه ها معمولاً از ریل اصلی خود خارج می شوند زیرا هدف اصلی خود را گم کرده و یا اصلا این هدف مشخص نیست.
۳۰) نداشتن تجربه و ارتباط مورد نیاز برای جذب سرمایه گذار و نبود مدیریت صحیح در مسائل مالی شرکت و پروژه
۳۱) **توهم تکنولوژی برتر** : برخی افراد تصور می کنند که اگر در یک پروژه از تکنولوژی های به اصطلاح جدیدتری استفاده کنند پروژه شکست نخواهد خورد این افراد به بیماری High-Tech illusion دچار هستند .
۳۲) انگیزش ضعیف تیم پروژه
۳۳) کار سخت (به جای کار هوشمندانه)
۳۴) برنامه ضعیف آموزش
۳۵) تغییرات سریع تکنولوژی
۳۶) فقدان و یا عدم کفایت مستندات
۳۷) ناتوانی روش های تولید نرم افزار در پاسخگویی به افزایش تقاضا
۳۸) عدم توجه به سند معماری سازمانی و سند حکمرانی داده
۳۹) تغییر استراتژیها و تاکتیکهای فناوری اطلاعات :
   a. با گذشت زمان
   b. ظهور ابزارهای جدید
   c. تغییر روندهای فناوری و تکنولوژی
   d. در زمان مواجه شدن با مشکلات و ریسکها

متاسفانه تجربیات بدست آمده از یک پروژه نرم افزاری، اثر بخشی مناسبی در پروژه های دیگری ندارد چرا که :

- پروژه‌های نرم‌افزاری از تنوع و دینامیک ساختاری بسیاری برخوردار است.
- در پروژه ها مشابه، دانش و تجربه کارشناسان کاملا متفاوت از سایر پروژه ها می باشد.
- نیازمندی‌ها پروژه ها دائما در حال تغییر است. همین مسئله نیازمند معماری و طراحی متفاوتی است.
- تغییرات سریع تکنولوژی، انتخاب‌های گوناگونی را پیش روی مدیران پروژه قرار میدهد که هر انتخاب ریسک مربوطه به خود را دارد.

علل فنی/تکنیکی شکست پروژه

۱) بی تجربگی و عدم تعهد کاری مدیران پروژه

۲) بی توجهی به اصول اولیه/پایه مدیریت پروژه[۲۷۹]

۳) عدم رعایت استانداردها و بهروشهای مهندسی نرم افزار

۴) **عدم درک درست از نیازمندیهای کاربران :** به سرعت قابل تشخیص و رفع شدن است.

۵) **معماری فنی نامناسب :** شناسائی این ریسک سخت بوده و معمولا تا به یک بن بست پیاده سازی نرسید متوجه چنین اشتباهی نخواهید شد.

۶) **اشتباه در طراحی :** از خطرناکترین اشتباهات بوده و تشخیص آن بسیار سخت است. چرا که معمولا تمام نیازهای عملیاتی کابران پوشش داده می شود ولی کاربران همچنان ناراضی است و پیاده سازی فعالیتها یکی پس از دیگری سخت تر شده و تغییرات هر بار با هزینه بیشتری همراه است.

۷) ضعف در فرآیند مهندسی معکوس و مهندسی مجدد

۸) Over-Engineering

۹) ایده آل گرائی افراطی

۱۰) ابزارگرائی

۱۱) برنامه ریزی نامناسب

۱۲) کمبود نیروی فنی متخصص و با تجربه

۱۳) عدم ثبات منابع انسانی در تیم پروژه

۱۴) عدم شناخت صحیح تکنولوژی مورد استفاده

۱۵) پشتیبانی اجرایی ضعیف

۱۶) عدم تحویل به موقع

۱۷) عدم تامین نیازمندی های کاربر

۱۸) کیفیت پایین و پایدار نبودن نرم افزار

۱۹) تاکید بیش از حد به استفاده از یک تکنولوژی خاص بدون در نظر گرفتن مزیتهای تکنولوژیهای جایگزین مشابه

۲۰) **سندرم عجله :** وقت برای فکر(مهندسی طراحی) کردن نیست ، فقط انجام بده **(نتیجه : بدهی فنی)**.

[279] - https://en.wikipedia.org/wiki/Project_management_triangle

پیوست ۱ : انواع معماریها

Event Driven Architecture (EDA)[۲۸۰]

Event-driven architecture **(EDA)** offers several key benefits, including Real-Time Responsiveness, improved scalability, enhanced resilience, and simplified integration with other systems. It achieves this by decoupling components, enabling asynchronous communication, and facilitating real-time event processing. Event-driven architecture (EDA) offers many benefits but also presents several disadvantages. Key drawbacks include increased complexity in design and implementation, challenges in debugging and troubleshooting due to asynchronous processing, and potential issues with maintaining data consistency across different components.

What does the term "event" mean in EDA?

An event is any change in state for system hardware or software. A change in state is called an event. In an event-driven architecture, everything that happens within and to your enterprise is an event, customer requests, inventory updates, sensor readings, and so on.
Event is a message that signifies something has occurred in the system. The event is sent by an Event Producer to an Event Broker, which then routes it to interested Event Consumers. Events allow the system to respond to changes asynchronously, enabling better scalability and flexibility.

With event-driven architecture, you can do the following:

- Everything happens as soon as possible, and nothing is waiting on anything else.
- You don’t have to consider what’s happening downstream, so you can add service instances to scale.
- Topic routing and filtering can divide up services quickly and easily, as in command query responsibility segregation.
- To add another service, you can just have it subscribe to an event and have it generate new events of its own; there’s no impact on existing services.

When to use EDA:

- **Parallel Processing and Fanout:** Enables a single event to trigger multiple processes simultaneously, without custom code for each consumer.
- **Cross-account/Cross-org and Cross-region Coordination and data Replication:** Ideal for systems that operate across different organizational units or geographical locations.
- **Integrating Diverse/Heterogeneous Systems:** Facilitates communication between systems with different underlying technologies without tight coupling.
- **Building Systems that require Audit Logging:** EDAs can simplify building audit logs as system communication can be logged as part of the messaging infrastructure.
- **Complex, distributed systems:** EDA is well-suited for applications with many interconnected components or microservices.
- **Systems requiring real-time updates:** When immediate responses to events are critical (e.g., fraud detection, financial transactions).
- **Systems with high data volume and throughput:** EDA can handle large amounts of events efficiently.
- **When decoupling and independent scalability are desired:** EDA promotes modularity and allows for independent scaling of services
- Resource state monitoring and alerting
- Enhanced Resilience and Accountability

Use Cases:

[280] - https://en.wikipedia.org/wiki/Event-driven_architecture , https://en.wikipedia.org/wiki/Staged_event-driven_architecture , https://www.geeksforgeeks.org/system-design/event-driven-architecture-system-design/
Book: Designing Event-Driven Systems By Ben Stopford, O’Reilly 2018

- Multiple subsystems must process the same events.
- Real-time Analytics/Processing with minimum time lag is required.
- Data Sharing and Democratization – Enabling multiple applications to access and utilize real-time data.
- Complex event processing, such as pattern matching or aggregation over time windows, is required.
- High volume and high velocity of data is required, such as with IoT.
- You need to decouple producers and consumers for independent scalability and reliability goals.
- Integrating applications
- Sharing and democratizing data across applications
- Connecting IoT devices for data ingestion and analytics
- Event-enabling microservices

Real-World Examples:

- E-Commerce Website:
  - E-commerce Order Processing
  - Inventory Management
  - Billing System
  - Realtime Notification System
- Smart Home System:
  - Security System
  - Lighting System
  - HVAC System
- Internet of Things (IoT) Data Collection
- User Registration & Authentication
- Stock Market and High Frequency Trading
- Real-Time Analytics
- Workflow Management[281]
- Sensor Integration in Smart Homes
- Event-Driven Microservices
- Online Multiplayer Games
- Live chat
- Distributed caching
- Distributed logging
- Digital twins[282]

Advantages and Benefits of EDA:

- Asynchronous messaging/communication, ideal for real-time processing and high-volume environments, offering high scalability and fault tolerance.
- Modularity (Decoupling and Loose Coupling): Producers and consumers are decoupled. Event-driven architecture encourages the creation of independently deployable services which can cause better code maintainability and easier team collaboration.
  - **Decoupling** is the practice of eliminating or minimizing direct dependencies between separate components within a system, so that no one component must rely on another. In the context of EDA, decoupling is achieved by making sure that the components generating events are sending event data without a particular consumer component in mind. This disconnection allows for components to be independent and creates a more flexible system overall.

[281] - https://en.wikipedia.org/wiki/Workflow , https://en.wikipedia.org/wiki/Workflow_management_system , https://github.com/meirwah/awesome-workflow-engines
[282] - https://en.wikipedia.org/wiki/Digital_twin

  - **Loose coupling** is a specific form of decoupling that aims to reduce the degree of interdependence between components, but not separate them completely. In a loosely coupled system, components may interact with one another but do so in a way that doesn't create any form of reliance.
- Highly Distributed and Scalable: It's highly scalable, elastic, and distributed.
- Real-time Processing/Workflows and Responsiveness: Consumers can respond to events immediately as they occur.
- Reliability and Improved Fault Tolerance: With decoupled components, failures in one part of the system are less likely to cascade and affect other parts.
- Seamless Integration with Disparate Systems
- There are no point-to-point integrations. It's easy to add new consumers to the system.
- Subsystems have independent views of the event stream.
- Event Sourcing: Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes. EDA can support event sourcing where all changes to an application's state are captured as a sequence of immutable events. This gives you a reliable audit trail and simplifies debugging.
- Flexibility and Agility: New components can be added or existing ones modified without major disruptions as long as they adhere to the event contract.
- Resilience: If one service fails, the system can continue processing other events. Events can also be stored and retried later, ensuring data isn't lost.
- Real-Time Processing: EDA is ideal for use cases that require real-time updates, such as notifications, order processing, or IoT data processing.
- Simplified Communication: The asynchronous nature of EDA means services don't need to directly interact with each other's APIs, reducing the complexity of synchronous communication protocols.
- Consumers and producers are separated.
- No point-to-point integrations. It is easy to add new consumers to the system.
- Subsystems have independent views of the event stream.

Disadvantages of EDA:

1. Complexity:

- **Increased system complexity:** EDA can be more complex to design, implement, and debug than simpler architectures. EDAs introduce additional components like event producers, brokers, and consumers, which can make the overall system more complex to understand and manage.
- **Eventual consistency:** Due to the asynchronous nature of EDA, data consistency might be eventual rather than immediate. Maintaining data consistency across all components can be difficult, as events might be processed at different times and in different orders.

2. Debugging and Troubleshooting:

- **Testing challenges:** Testing event-driven systems can be more difficult due to the distributed and asynchronous nature.
- **Difficult Debugging:** Tracing the flow of events in an EDA can be challenging, especially in distributed systems, as events can trigger actions across multiple services asynchronously.
- **Tracing Issues:** It can be difficult to trace the origin and path of an event, making it hard to pinpoint the source of a problem.
- **Complex Error Handling:** Handling errors and exceptions can be tricky, as an event might trigger a cascade of actions across different services.

3. Cost and Monitoring:

- **Higher Infrastructure Costs:** EDAs may require more infrastructure to handle event management and message queuing.
- **Complex Monitoring:** Monitoring event-driven systems can be more challenging, requiring specialized tools and techniques to track event flows and identify performance bottlenecks.

4. Other Challenges:

- **Event Ordering:** Maintaining the order of events can be problematic, especially in distributed systems where events might arrive out of sequence. Ensuring the correct order of events can be challenging in distributed systems. Some events may arrive out of sequence or with delays which require careful handling.
- **Overhead:** While offering decoupling, EDA can introduce overhead in terms of event handling and message routing.
- **Skill Gap:** There might be a shortage of developers with the necessary experience in building and managing event-driven systems.
- **Operational overhead:** Managing event brokers, message queues, and event routing can add operational complexity.
- **Lack of Context:** Unlike request-response systems, EDA lacks explicit context propagation, making it harder to understand the full picture of a user request's journey.
- **Latency:** In some cases, event-driven systems may introduce latency because of the asynchronous nature of event processing.
- **Message Broker Reliability:** Event-driven architecture often relies on message brokers for event distribution. The reliability and scalability of these brokers become crucial points of failure.
- **Delivery Guarantees**
    - **Almost once:** The producer transfers a message, and the consumer demand may / may not receive it.
    - **At least once:** The producer transfers a message, and the consumer may prepare duplicate occurrences of the message.
    - **Exactly once:** The producer transfers a message exactly once, and the consumer processes it exactly once in EDA.

Principles of Event-Driven Architecture:

- Use a network of event brokers to make sure the right “things” get the right events.
- Use topics to make sure you only send once and only receive what you need.
- Use an event portal to design, document, and govern event-driven architecture across internal and external teams.
- Use event broker persistence to allow consumers to process events when they’re ready (deferred execution).
- Remember, this means not everything is up-to-date (**eventual consistency**).
- Use topics again to separate out different parts of a service (command query responsibility segregation).

Event-driven architecture models/patterns

- **Publish/subscribe (aka “pub/sub”)**: With pub/sub, event consumers subscribe to messages and channels published by event producers. When an event is published, it is sent directly to all subscribers via a broker. To avoid duplication, events cannot be replayed or accessed once consumed, they are deleted by the broker.
- **Event streaming**: With event streaming, producers publish entire streams of events to a broker. Consumers subscribe to the stream and can read from any part it, consuming only the events that are relevant to them. With this pattern, events are retained by the broker even after they are consumed.

Patterns of event processing

- **Simple event processing:** Consumers process each event as it is received.
- **Complex event processing** (CEP)**:** Consumers process a series of events to detect patterns and perform actions based on the result.
- **Event stream processing:** Consumers process and act on a constant flow of data (data in motion) in real time using a data streaming platform.

- Event Sourcing:
- CQRS (Command Query Responsibility Segregation)

Implementing EDA

To create and event driven system basic steps are like following:

- **Identify Events:** Determine the key events that are significant in your system.
- **Define Event Producers:** Identify the components responsible for generating these events.
- **Select Event Channel:** Choose an appropriate event channel or messaging system, such as Apache Kafka or RabbitMQ.
- **Define Event Consumers:** Identify and configure the components that will consume and process the events.
- **Testing and Monitoring:** Thoroughly test your system to ensure events are correctly processed. Use monitoring tools to observe system performance.

Best Practices for Implementing EDA

- Event Granularity:
  - Coarse-Grained Events
  - Fine-Grained Events
- Idempotency:
  - Idempotent Operations
  - Unique Identifiers
- Error Handling:
  - Retry Mechanisms
  - Dead Letter Queues
- Scalability Considerations:
  - Load Balancing
  - Partitioning:

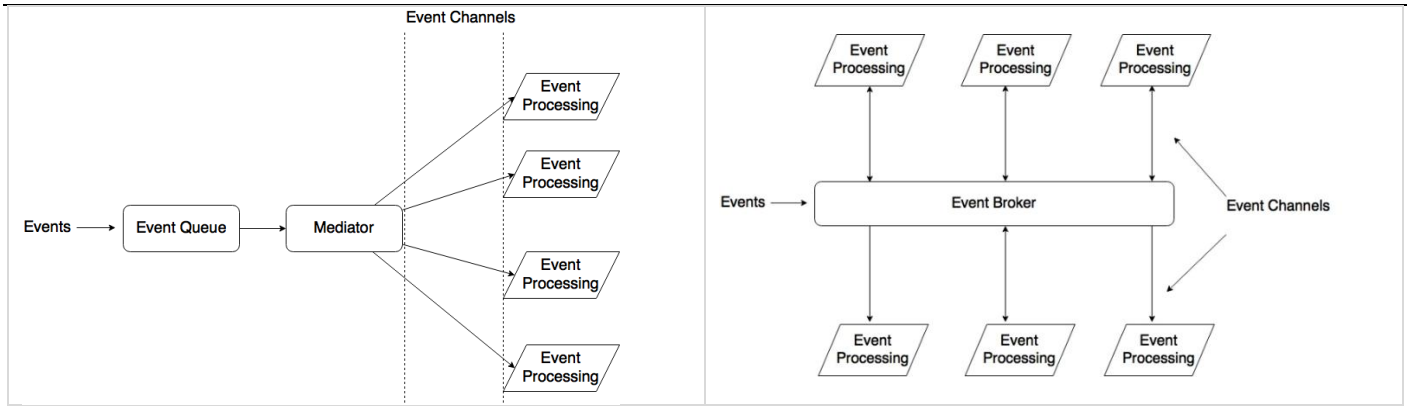
Components of Event-Driven Architecture

- Events
- Event APIs
- Event Bridge
- Event Types:
  - Domain Events: These events reflect significant changes or actions in your business domain.
  - Integration Events: These are events that are designed to trigger actions in other systems or services.
  - External Events: Events that originate outside the system, like a payment confirmation from a third-party service.
  - System Events: These events represent system-specific information, often related to the state of the system.
  - Notification Events: These events serve the purpose of notifying consumers about something that happened.
- Event Catalog
- Event Channels
  - Pub/Sub (Publish-Subscribe)
    - Dashboards
    - Interactive learning
  - Event streams
    - IoT
    - Microservices
- Event Producers/Source (Publisher)

- Event Brokers/Bus (Message Broker)
- Event Choreography
- Event Consumer/Processor (Subscriber)
- Envet Delivery
    - Acknowledgements
    - Dead Letter Queue and Retry
    - Transactional Messaging
- Event-Driven Patterns
- Event Flow
- Event Gateway
- Event Generator (for testing)
- Event Governance
- Event History
- Error Handling & Retry Mechanisms
- Event Log
    - Distributed Tracing (Jaeger and Zipkin)
    - Centralized Logging (ELK Stack)
    - Metrics and Alerts (Prometheus)
- Event Mesh
- Event Ordering
    - Partitioning
    - Sequence Number
    - Idempotency
- Event Portal
- Event-Processing Patterns
    - Real-Time Processing
    - Event Replay
    - Eventual Consistency
- Event Queues
- Event Router
- Event Schema Registry
- Event Stores: EventStoreDB: A database specifically designed to store events, often used with event sourcing.
- Event Streaming
- State Managements
    - Event Sourcing
    - CQRS

EDA Topologies

- Mediator Topology
- Broker Topology



How Is Event Driven Architecture Used for Streaming Data?

There are two main kinds of event-driven architectures: **message queues and stream processing**. Of the two, streaming is the more asynchronous and decoupled model.

Data streaming is a processing model supported by event driven architectures. Unlike traditional message queues, data streaming completely decouples event products from event consumers. The result is a fully asynchronous messaging infrastructure that permits great flexibility and responsiveness. It also prevents performance bottlenecks that might occur in more tightly-coupled systems.

Stream processing has emerged as the leading choice for most event-driven systems. The primary benefit over message queues is the dramatic increase in flexibility, as streaming supports multiple consumers, event replay, and sliding window statistics.

What Is Good Event Driven Architecture Tools?

The most popular event processing processing platform today is **Apache Kafka**, a distributed data streaming platform. Apache Kafka is a distributed streaming platform that supports a topic-based producer-consumer communication pattern that combines benefits of both message queues and publish-subscriber patterns. Kafka supports real-time publishing, subscribing, storing, and processing of event streams, and is generally used for applications that require high throughput and scalability. A key benefit of Kafka is the elimination of point-to-point integrations for data sharing in certain applications.

How to Choose a Database for Event Driven Architecture

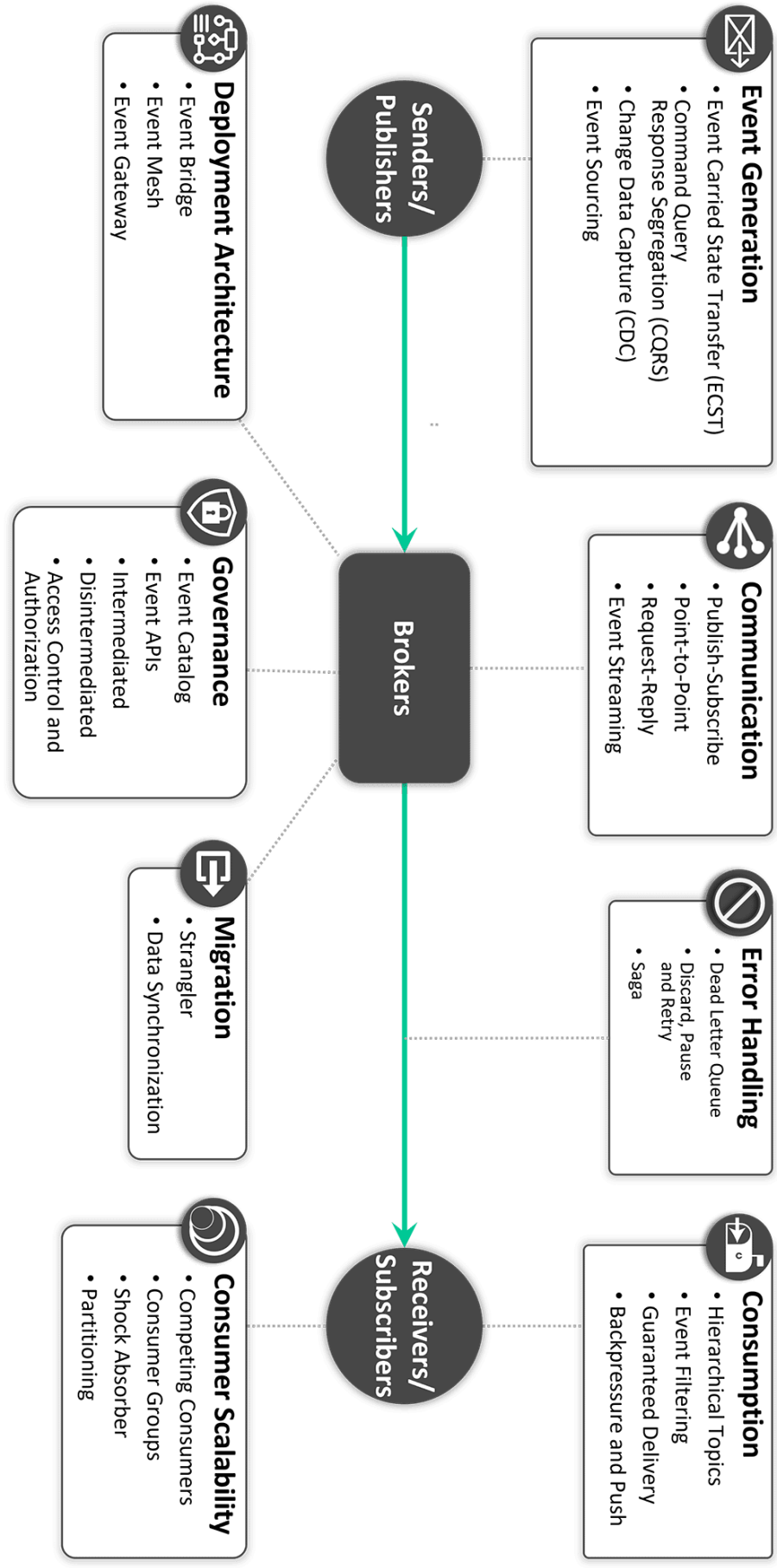
As noted, Apache Kafka is architected to support trillions of events per day with real-time responsiveness. Very few databases available today are capable of supporting these requirements. Traditional relational database management systems (RDBMS) are simply built for a different application paradigm. **NoSQL** databases, which offer high availability and scale-out capabilities are better suited for event driven architectures. However, even within the expansive family of NoSQL databases, only very few are capable of supporting the extreme requirements imposed by event driven architecture. The overall performance of event driven data architecture is often bottlenecked by the performance of the underlying data storage layer. **Apache Cassandra and ScyllaDB** are ideal backend for event-driven architecture.

Event-Driven Architecture Patterns[283]

- Event Generation Patterns:
  - Event Carried State Transfer (ECST)
  - Command Query Response Segregation (CQRS)
  - Change Data Capture (CDC)
  - Event Sourcing
- Communication Patterns:
  - point-to-point
  - publish/subscribe
  - request/reply

283 - https://solace.com/event-driven-architecture-patterns/

    - event streaming
- Consumption Patterns:
    - hierarchical topics
    - event filtering
    - guaranteed delivery
    - backpressure and push
    - exclusive consumer
- Consumer Scalability:
    - competing consumers
    - consumer groups
    - shock absorber
    - partitioning
- Deployment Architecture Patterns:
    - event bridge
    - event mesh
    - event gateway
- Error Handling Patterns:
    - dead letter queue
    - discard, pause and retry
    - saga
- Governance Patterns:
    - event catalog
    - event APIs
    - Intermediated Governance
    - Disintermediated Governance
    - access control and authorization
- Migration Patterns:
    - Strangler
    - data synchronization


Event Generation
• Event Carried State Transfer (ECST)
• Command Query Response Segregation (CQRS)
• Change Data Capture (CDC)
• Event Sourcing
Senders/ Publishers
Deployment Architecture
• Event Bridge
• Event Mesh
• Event Gateway
Communication
• Publish-Subscribe
• Point-to-Point
• Request-Reply
• Event Streaming
Brokers
Governance
• Event Catalog
• Event APIs
• Intermediated
• Disintermediated
• Access Control and Authorization
Error Handling
• Dead Letter Queue
• Discard, Pause and Retry
• Saga
Migration
• Strangler
• Data Synchronization
Receivers/ Subscribers
Consumption
• Hierarchical Topics
• Event Filtering
• Guaranteed Delivery
• Backpressure and Push
Consumer Scalability
• Competing Consumers
• Consumer Groups
• Shock Absorber
• Partitioning

MicroServices (Domain Actors) [284]

Microservices is an architectural style that structures an application as a collection of small, independent, and loosely coupled services. Each service is responsible for a specific business capability and can be developed, deployed, and scaled independently. Microservices typically communicate with each other using lightweight protocols, such as **RESTful APIs (synchronous) or message queues/event streams (asynchronous).** Microservices architecture is a powerful approach to building scalable, resilient, and maintainable applications, but it requires careful planning and management to overcome the inherent complexities of distributed systems.

Here's a more detailed breakdown:[285]

- **Independent Services:** Each microservice is a self-contained unit responsible for a specific business capability, like user authentication, order processing, or product management.
- **Loose Coupling:** Microservices are designed to minimize dependencies between them, allowing for independent development, deployment, and scaling.
- **Lightweight Communication:** Services communicate through lightweight protocols, often REST APIs, enabling efficient and flexible interactions.
- **Independent Deployments:** Microservices can be deployed and updated independently without affecting other services or the overall application.
- **Decentralized Data Management:** Each microservice typically manages its own data, promoting autonomy and preventing tight coupling through shared databases.
- **Fault Isolation:** If one microservice fails, it shouldn't necessarily bring down the entire application, as other services can continue to operate independently.
- **Technology Diversity:** Teams can choose the best technology stack for each microservice, potentially using different programming languages, frameworks, and databases.
- **Future-proofing & Backward compatibility:** Optimization, Switching technology, Adding features, Scaling at the service level. Backward compatibility is a key strategy for future-proofing.

Note:

1) Patterns[286]
2) SAGA[287] distributed transactions pattern
    a. Choreography[288]
    b. Orchestration
3) API Gateway
4) API-Driven Communication

Use Cases:

Microservices are well-suited for:

- **Large, complex applications:** Where the benefits of modularity and independent development outweigh the complexities.
- **Applications with frequent updates and deployments:** Enabling faster release cycles and continuous delivery.
- **Applications with diverse technology requirements:** Allowing teams to choose the best technologies for each service.
- **Applications that require high scalability and availability:** Enables scaling individual services based on demand and fault isolation.

Microservices Cons

1. Operational Complexity

Microservices are flexible and scalable yet complicated to manage. Managing dozens or hundreds of services with their own codebases, deployment processes, and databases is difficult. Strong infrastructure and tooling are needed to keep services safe, updated, and communicating. Distributed systems are harder to monitor and debug than monolithic designs since faults in one service might affect others.

---

284 - https://en.wikipedia.org/wiki/Microservices , https://microservices.io/
https://microservices.io/patterns/microservices.html , https://eventuate.io/exampleapps.html
https://cloud.google.com/learn/what-is-microservices-architecture

285 - https://azure.microsoft.com/en-us/blog/microservices-an-application-revolution-powered-by-the-cloud/
https://www.martinfowler.com/articles/microservices.html
https://martinfowler.com/bliki/MicroservicePrerequisites.html
https://www.infoq.com/articles/CCC-Jimmy-Nilsson/

286 - https://microservices.io/patterns

287 - https://microservices.io/patterns/data/saga.html
https://learn.microsoft.com/en-us/dotnet/architecture/microservices
https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga
https://dotnet.microsoft.com/en-us/download/e-book/microservices-architecture/pdf

288 - https://learn.microsoft.com/en-us/azure/architecture/patterns/choreography

2. Consistent Data and distributed data management
Data consistency between microservices is difficult. Each service handles its own data store, therefore maintaining it in sync across different services might require sophisticated patterns like sagas (local transactions that assure consistency) or event sourcing (recording all application state changes as events). These methods need careful preparation and increase system complexity. If high availability and real-time updates are needed, synchronizing the inventory and order processing services on an e-commerce platform might be problematic.

Microservices Challenges and Pitfalls:

- **Increased Complexity:** Managing a distributed system with multiple services adds complexity to development, deployment, and monitoring.
- **Inter-Service Communication Overhead:** Communication between services can introduce latency and potential bottlenecks.
- **Data Consistency:** Ensuring data consistency across multiple databases can be challenging.
- **Distributed Transactions:** Implementing distributed transactions across multiple services can be complex.
- **Observability:** Monitoring and debugging a distributed system requires robust tooling and strategies.
- **Extra work:** Servie contract maintenance, Handling service failures, …
- **Sharing functionalities, features and dependencies:**
- **Cocktail of technologies:**[289]

Microservices best practices

When we develop microservices, we need to follow the **best practices**:

1) Use separate data storage for each microservice
2) Keep code at a similar level of maturity
3) Separate build for each microservice
4) Assign each microservice with a single responsibility
5) Deploy into containers
6) Design stateless services
7) Adopt domain-driven design
8) Design micro frontend
9) Orchestrating microservices

Understanding Event-Based Architecture:

Event-based architecture is centered around the idea of loosely coupling services through the exchange of events. In this model, services produce events when specific actions or changes occur. Other services can subscribe to these events and react accordingly. The key features of event-based architecture include asynchronous communication, event-driven workflows, and eventual consistency.

When to Choose Event-Based Architecture:

- **Scalability and Resilience:** Event-driven systems are highly scalable as services can process events independently. If your application requires handling many events or needs to scale dynamically, event-based architecture can be a good fit.
- **Loose Coupling:** Event-driven architecture promotes loose coupling between services. This decoupling allows services to evolve independently, making it easier to introduce new services or modify existing ones without affecting the entire system.
- **Real-time Updates and Reactiveness:** If your application demands real-time updates or reactive behavior, event-driven architecture is well-suited. Events can be instantly propagated, enabling quick responses to changes and ensuring data consistency across the system.
- **Event Sourcing and CQRS:** Event-based architecture is a natural fit for event sourcing and Command Query Responsibility Segregation (CQRS). Event sourcing captures all changes to an application state as a sequence of events, while CQRS separates the read and write models. These patterns can be advantageous for certain domains where auditability, scalability, or complex data processing is required.

Understanding Request-Based Architecture:

Request-based architecture follows a more traditional client-server model, where services communicate through synchronous requests and responses. Each service exposes a well-defined API, and other services invoke these APIs to access functionality or request data. This architecture emphasizes simplicity, ease of development, and strong consistency.

[289] - https://ar5iv.labs.arxiv.org/html/2203.07267 (Development Frameworks for Microservice-based Applications: Evaluation and Comparison)

When to Choose Request-Based Architecture:

- **Simplicity and Ease of Development:** Request-based architecture offers a simpler development model as services interact through well-defined APIs. This makes it easier to design, implement, and test individual services, especially for applications with straightforward requirements.
- **Strong Consistency:** If your application demands strict consistency requirements, such as maintaining data integrity across multiple services within a transactional context, request-based architecture can provide the necessary control and guarantees.
- **Point-to-Point Communication:** Request-based architecture simplifies communication patterns, especially when a service needs to directly call another service for a specific task. This approach is often suitable for systems with limited or predefined communication flows.
- **Existing Infrastructure and Tools:** If you already have a mature infrastructure or tooling that is better suited for request-based communication, such as API gateways or service meshes, it may be more practical to stick with request-based architecture.

Scaling Microservices: Message-Driven vs Event-Driven Architecture

Before diving into message-driven and event-driven architectures, it's essential to understand the key concepts of messages and events.

- **Message:** A message is a piece of data sent from one service to another to perform a specific task or action. It is directed to a particular recipient with a clear intent (e.g., "process this order" or "update this record").
- **Event:** An event represents a fact about something that has occurred in the system. It describes a past occurrence (e.g., "an order was placed") and is broadcast to interested consumers without a specific recipient in mind.

Message-Driven Architecture

Message-driven architecture focuses on the direct exchange of messages between services. It excels in task-based workflows where clear communication and acknowledgment are crucial. Message-driven architecture is ideal for **workflows** that need sequential execution and task acknowledgment. However, it may not be the best choice for scenarios requiring **real-time** processing or high **parallelism**. Best for task-oriented workflows where reliability, sequential processing, and simplicity are essential.

Key Features:

- **Point-to-point communication:** Messages are sent to specific recipients.
- **Reliability:** Acknowledgment mechanisms ensure reliable task completion.
- **Sequential processing:** Tasks are executed in a defined sequence.

Use cases:

- **Task-based workflows:** Ideal when sequential task processing is required.
- **Reliability:** Best for systems needing task acknowledgment and assurance of task completion.
- **Direct communication:** Use when explicit, point-to-point communication between services is critical.

Event-Driven Architecture

Event-driven architecture centers on state changes and their reactions. When an event occurs (e.g., "OrderPlaced"), it is broadcast to multiple consumers who react independently, **making it a great fit for real-time data processing and parallel workflows**. Event-driven architecture is excellent for **scalability and loose coupling**, but it can introduce complexities, such as managing eventual consistency and cascading dependencies. Ideal for real-time processing, high scalability, and loosely coupled systems.

Key Features:

- **Decoupled communication:** Producers emit events without knowing the consumers, allowing flexible reactions.
- **Parallel processing:** Multiple services can react to the same event simultaneously.
- **Historical state tracking:** Events can be replayed to reconstruct system state or analyze past data.

Use cases:

- **Real-time data processing:** Best for scenarios involving high-frequency data and real-time reactions.
- **Parallel workflows:** Perfect for scaling and decoupling services that react to events simultaneously.
- **Historical tracking:** Necessary for systems that need to rebuild state or analyze past data.

Key technologies supporting microservices

- **Containers (e.g., Docker):**
- **Orchestration platforms (e.g., Kubernetes):**
- **Service mesh (e.g., Istio):** A service mesh enhances communication between microservices, providing features like traffic management, security, and observability. It acts as a dedicated infrastructure layer for inter-service communication, improving resilience and reducing development overhead.
- **Serverless computing (e.g., AWS Lambda):** Serverless platforms abstract away infrastructure management, allowing developers to focus solely on code. This model can be highly cost-effective for microservices, as resources are consumed only when needed, and scaling infrastructure like this is seamless.

Anti Patterns for Microservices Architeture

Learning anti-patterns in microservices is crucial for avoiding common mistakes. Below are some anti-patterns in microservices and by understanding these anti-patterns, developers can make informed decisions and implement best practices.

- **Shared database:** Coupling between services. When microservices share a single centralized database, it can compromise their independence and scalability.
- **Too fine-grained services:** Leads to performance issues. Microservices that frequently communicate for minor tasks can create excessive network traffic, leading to delays and increased latency.
- **Over-engineering:** Adding microservices unnecessarily. Creating too many microservices for small functions can add unnecessary complexity to the system.
- If the boundaries between microservices are not clearly defined, it can cause confusion about their responsibilities.
- Failing to address security issues in microservices can expose the system to vulnerabilities and potential data breaches.

10 Tips for failing at microservices (David Schmitz)[290]

1) Use A HR Driven Microservices Architecture
2) The Monolithic Business
3) The Decision Monolith
4) The Single Page Application (SPA) Monolith
5) The Distributed Monolith
6) Use The Meat Cloud
7) The Homegrown Monolith
8) The Event Monolith
9) The Data Monolith
10) Go Full-Scale Polyglot

Main components of Microservices Architecture

- **Microservices:** Small, loosely coupled services that handle specific business functions, each focusing on a distinct capability.
- **API Gateway:** Acts as a central entry point for external clients also they manage requests, authentication and route the requests to the appropriate microservice.
- **Service Registry and Service Discovery:** Keeps track of the locations and addresses of all microservices, enabling them to locate and communicate with each other dynamically.
- **Load Balancer:** Distributes incoming traffic across multiple service instances and prevent any of the microservice from being overwhelmed.
- **Containerization:** Docker encapsulate microservices and their dependencies and orchestration tools like Kubernetes manage their deployment and scaling.
- **Event Bus/Message Broker:** Facilitates communication between microservices, allowing pub/sub asynchronous interaction of events between components/microservices.
- **Database per Microservice:** Each microservice usually has its own database, promoting data autonomy and allowing for independent management and scaling.
- **Caching:** Cache stores frequently accessed data close to the microservice which improved performance by reducing the repetitive queries.
- **Fault Tolerance and Resilience Components:** Components like circuit breakers and retry mechanisms ensure that the system can handle failures gracefully, maintaining overall functionality.

Real-World Example of Microservices (Amazon E-Commerce Application)

- **User Service:** Handles user accounts and preferences, making sure each person has a personalized experience.
- **Search Service:** Helps users find products quickly by organizing and indexing product information.
- **Catalog Service:** Manages the product listings, ensuring all details are accurate and easy to access.
- **Cart Service:** Lets users add, remove, or change items in their shopping cart before checking out.
- **Wishlist Service:** Allows users to save items for later, helping them keep track of products they want.
- **Order Taking Service:** Processes customer orders, checking availability and validating details.
- **Order Processing Service:** Oversees the entire fulfillment process, working with inventory and shipping to get orders delivered.
- **Payment Service:** Manages secure transactions and keeps track of payment details.
- **Logistics Service:** Coordinates everything related to delivery, including shipping costs and tracking.
- **Warehouse Service:** Keeps an eye on inventory levels and helps with restocking when needed.
- **Notification Service:** Sends updates to users about their orders and any special offers.
- **Recommendation Service:** Suggests products to users based on their browsing and purchase history
- Shipping Service:
- Inventory Service:

[290] - https://speakerdeck.com/koenighotze/10-tips-for-failing-at-microservices

- Account Service:

Autopilot Microservices Principle [291]

Microservices should be architected to run on autopilot in their deployment environment. An autopilot microservice means a microservice that runs in a deployment environment without human interaction, except in abnormal situations when the microservice should generate an alert to indicate that human intervention is required.

Autopilot microservices principle requires that the following sub-principles are followed:

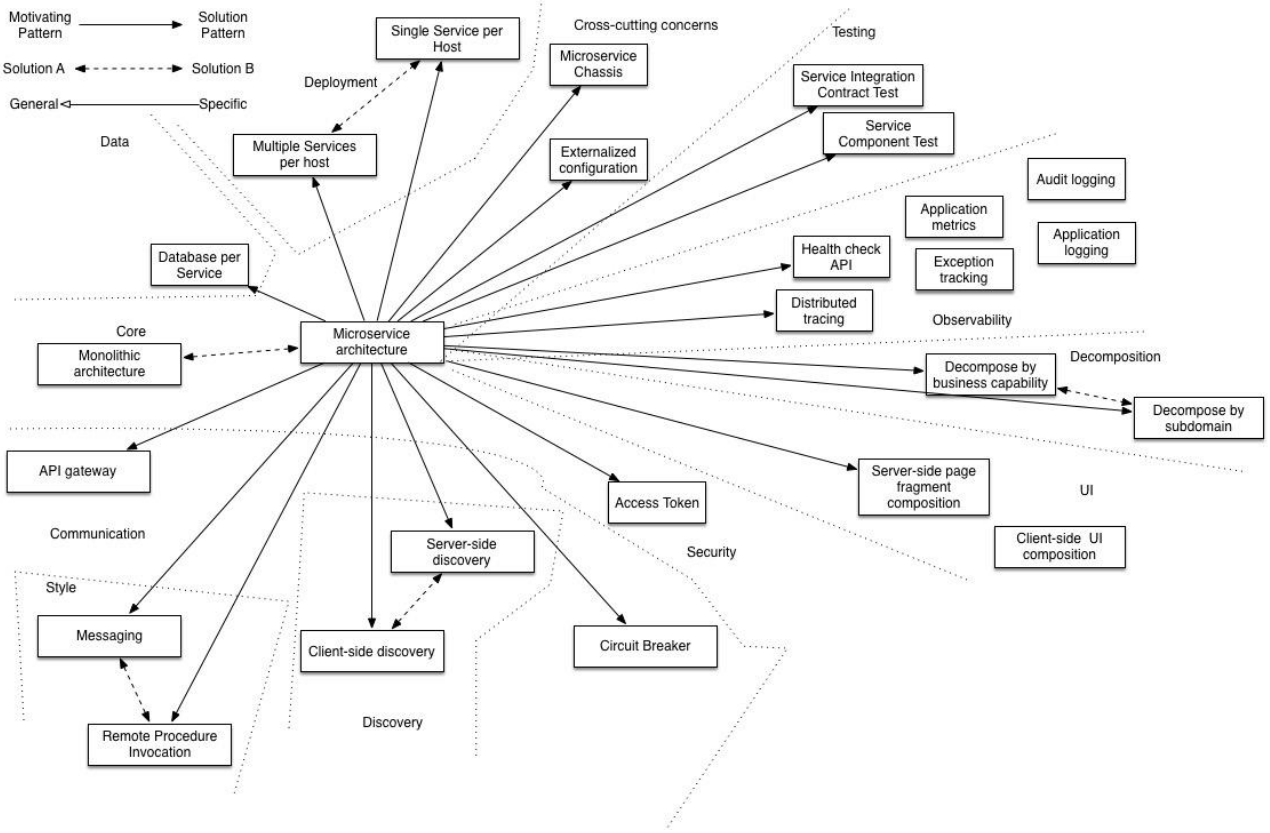
1) **Stateless microservices principle:** Microservices should be stateless to enable resiliency, horizontal scalability, and high availability. A microservice can be made stateless by storing its state outside itself. The state can be stored in a data store that microservice instances share. Typically, the data store is a database or an in-memory cache (like Redis, for example).
2) **Resilient microservices principle:** Microservices should be resilient, i.e., quickly recover from failures automatically. In a Kubernetes cluster, the resiliency of a microservice is handled by the Kubernetes control plane. If the computing node where a microservice instance is located needs to be decommissioned, Kubernetes will create a new instance of the microservice on another computing node and then evict the microservice from the node to be decommissioned.
3) **Horizontally autoscaling microservices principle:** Microservices should automatically scale horizontally to be able to serve more requests. Horizontal scaling means adding new instances or removing instances of a microservice. Horizontal scaling of a microservice requires statelessness. Stateful services are usually implemented using **sticky sessions** so that requests from a particular client go to the same service instance. The horizontal scaling of stateful services is complicated because a client's state is stored on a single service instance. In the cloud-native world, we want to ensure even load distribution between microservice instances and target a request to any available microservice instance for processing.
    a. Microservice must be stateless
    b. There must be one or more metrics that define the scaling behavior
4) **Highly-available microservices principle:** Business-critical microservices must be highly available.
5) **Observable microservices principle:** It should be possible to detect any abnormal behavior in deployed microservices immediately. Abnormal behavior should trigger an alert. The deployment environment should offer aids for troubleshooting abnormal behavior.

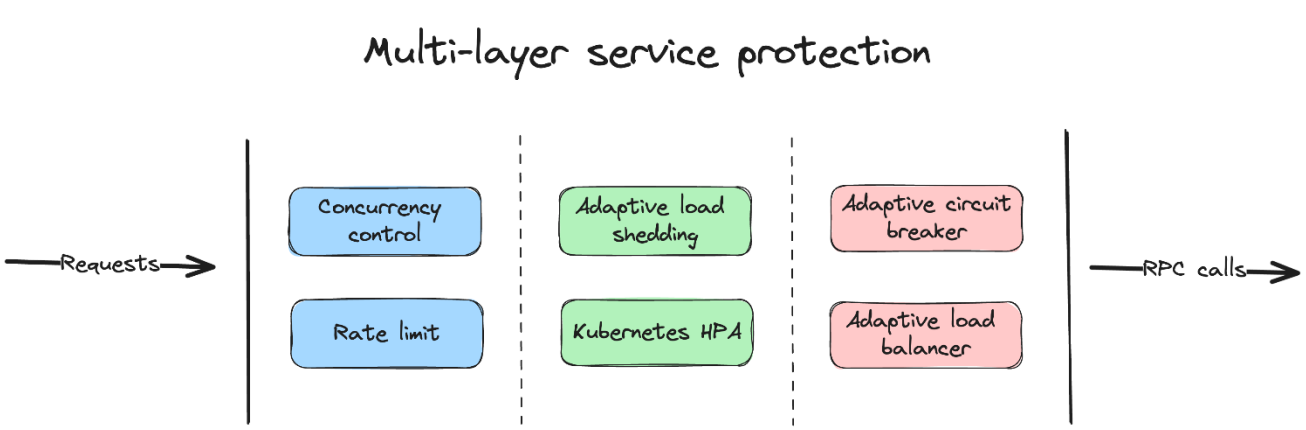
Microservices Design Patterns[292]

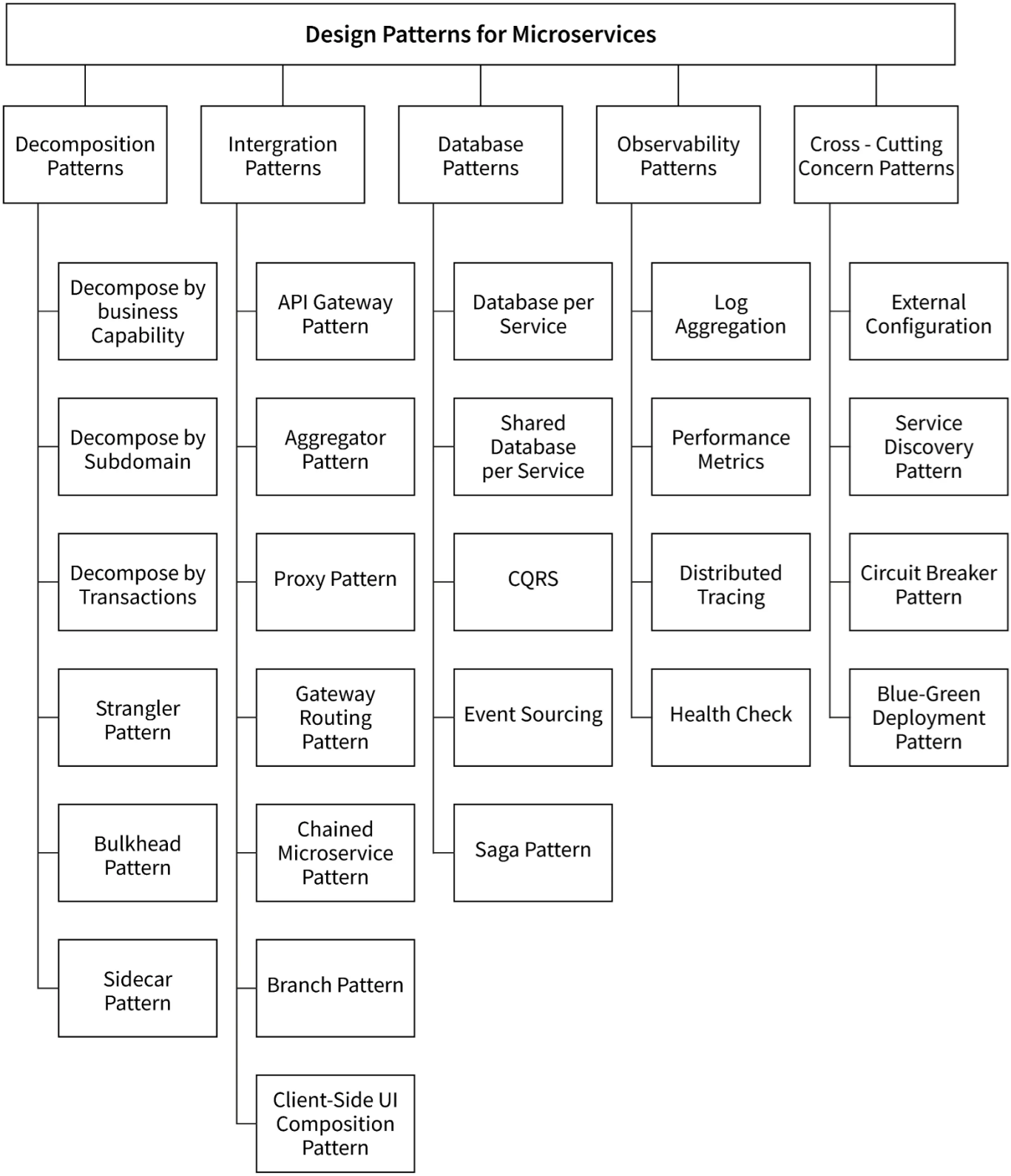
Microservices design patterns are created to mitigate the challenges of microservice architectures. By breaking applications down into small autonomous services, microservices enable greater flexibility, innovation, and performance than monolithic legacy systems. However, they can also increase the risks of overcomplicated infrastructures and inconsistencies in data between services.

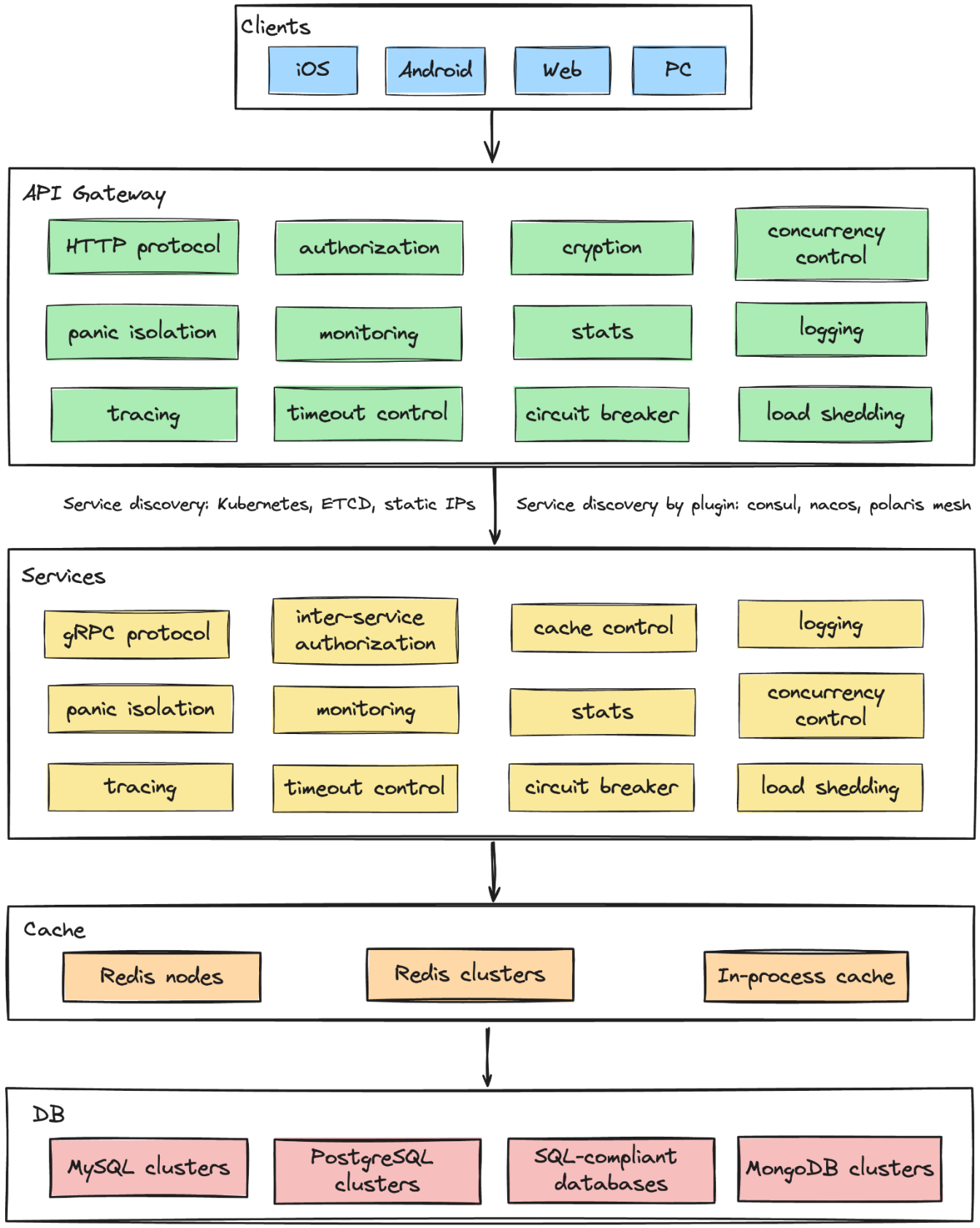
[291] - Clean Code Principles and Patterns Python Edition (Petri Silen, 2024) http://leanpub.com/cleancodeprinciplesandpatternspythonedition

[292] - https://microservices.io/patterns/microservices.html , https://learn.microsoft.com/en-us/azure/architecture/microservices/design/patterns https://learn.microsoft.com/en-us/azure/architecture/patterns/#pattern-catalog


Motivating Pattern
Solution Pattern
Solution A
Solution B
General
Specific
Data
Deployment
Single Service per Host
Multiple Services per host
Cross-cutting concerns
Microservice Chassis
Externalized configuration
Testing
Service Integration Contract Test
Service Component Test
Audit logging
Application metrics
Application logging
Exception tracking
Health check API
Distributed tracing
Observability
Database per Service
Core
Monolithic architecture
Microservice architecture
Decomposition
Decompose by business capability
Decompose by subdomain
API gateway
Communication
Style
Messaging
Remote Procedure Invocation
Server-side discovery
Client-side discovery
Discovery
Access Token
Security
Circuit Breaker
Server-side page fragment composition
UI
Client-side UI composition

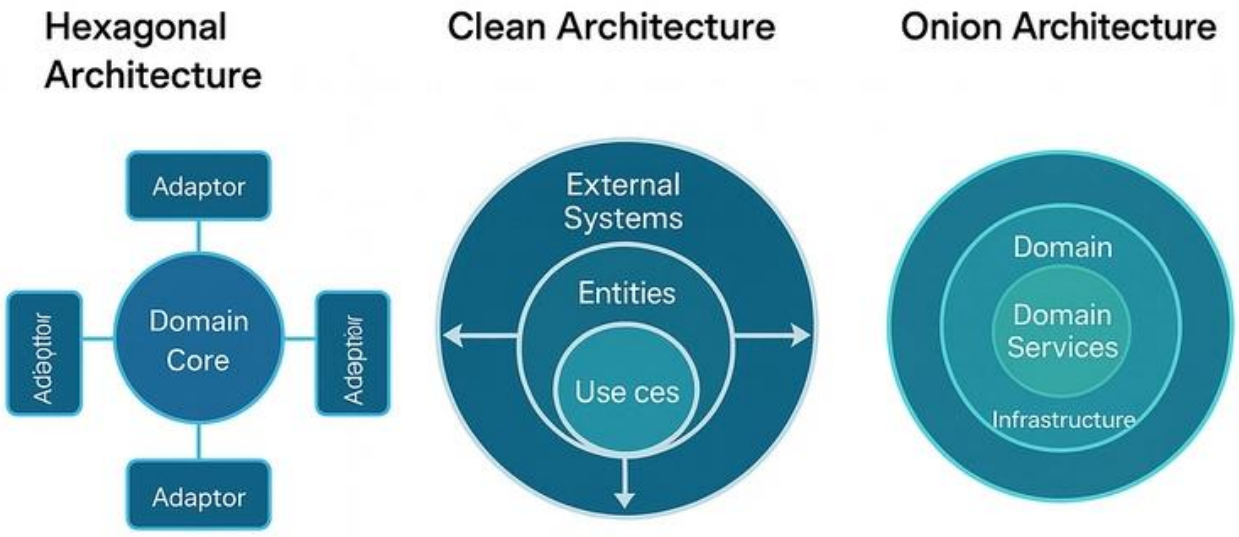

Multi-layer service protection
Requests
Concurrency control
Rate limit
Adaptive load shedding
Kubernetes HPA
Adaptive circuit breaker
Adaptive load balancer
RPC calls


Design Patterns for Microservices
Decomposition Patterns
Intergration Patterns
Database Patterns
Observability Patterns
Cross - Cutting Concern Patterns
Decompose by business Capability
Decompose by Subdomain
Decompose by Transactions
Strangler Pattern
Bulkhead Pattern
Sidecar Pattern
API Gateway Pattern
Aggregator Pattern
Proxy Pattern
Gateway Routing Pattern
Chained Microservice Pattern
Branch Pattern
Client-Side UI Composition Pattern
Database per Service
Shared Database per Service
CQRS
Event Sourcing
Saga Pattern
Log Aggregation
Performance Metrics
Distributed Tracing
Health Check
External Configuration
Service Discovery Pattern
Circuit Breaker Pattern
Blue-Green Deployment Pattern


Clients
iOS
Android
Web
PC
API Gateway
HTTP protocol
authorization
cryption
concurrency control
panic isolation
monitoring
stats
logging
tracing
timeout control
circuit breaker
load shedding
Service discovery: Kubernetes, ETCD, static IPs
Service discovery by plugin: consul, nacos, polaris mesh
Services
gRPC protocol
inter-service authorization
cache control
logging
panic isolation
monitoring
stats
concurrency control
tracing
timeout control
circuit breaker
load shedding
Cache
Redis nodes
Redis clusters
In-process cache
DB
MySQL clusters
PostgreSQL clusters
SQL-compliant databases
MongoDB clusters

Hexagonal vs Onion vs Clean

Once you dive into architecture discussions, you'll quickly notice that Clean, Onion, and Hexagonal often get lumped together. They all share the same DNA: keep business rules at the center, shove frameworks and databases to the edges. But their vibes and how painful they feel to implement are different.



Architecture	Complexity	Learning curve	Best fit	Vibe check
Clean	High	Steep	Enterprise-scale, multi-team	Strict, rules-heavy
Onion	Medium-High	Moderate	Domain-driven monoliths, mid-to-large apps	Practical, structured
Hexagonal	Medium	Beginner-friendly	Microservices, apps with multiple entry points	Pragmatic, flexible

Hexagonal architecture (aka Ports and Adapters)

The idea is simple: keep your business logic in the center, and push everything else databases, APIs, UIs to the edges. Think of your domain as the socket on the wall. The outside world connects through adapters: a SQL adapter, a NoSQL adapter, a REST API adapter, even a CLI adapter. The socket doesn't care what's plugged in, as long as it fits the port.

When (and when not) to use Hexagonal

So when should you actually go full Hex? Short answer: when you care about testability, churn survival, or juggling multiple entry points. Long answer: let's break it down.

When Hex makes sense

- You need testability. Writing clean unit tests without dragging in a real DB or API becomes way easier. Mocking a port is cleaner than mocking half your framework.
- You expect churn. Switching from SQL to NoSQL, from REST to gRPC, or from one framework to another? Ports and adapters are basically insurance.
- You're scaling. If your system has multiple ways in (APIs, queues, CLI tools), Hex keeps your domain logic from becoming a traffic jam of spaghetti.

When to skip it

- Tiny MVPs. If you've got fewer than five endpoints, Hex is overkill. Use KISS (Keep It Simple, Stupid) and just ship.
- Solo side projects. Unless you're secretly planning for unicorn status, you'll move faster without layers of ports and adapters.
- Short-lived apps. Hackathon project? Startup experiment? Don't waste cycles future-proofing something that might not see next month.

Onion architecture (Layered Domain-Centric Design)

Onion is like Clean's less preachy cousin. Still has rings, still emphasizes the domain model in the center, but it focuses more on layering your domain and application logic around that core. It's structured, but usually has less "thou shalt not" energy. If Clean feels academic, Onion feels more practical.

Clean architecture (A Flexible Framework-Agnostic Approach)

Uncle Bob's classic. Picture concentric circles where dependencies always point inward. The innermost ring is your use cases and entities. As you move outward, you hit interfaces, then frameworks and drivers. The rules are strict: nothing in the core should know about anything on the outside. It's powerful, but it can feel like playing a coding version of Dark Souls lots of rules, lots of ceremony.

Key Differences Between Onion Architecture and Clean Architecture

Layering and Dependency:

- Onion: Enforces strict layering, where each layer depends only on the layer directly inside it.
- Clean: More flexible with layering; you can skip layers and still keep the core logic independent.

Domain Focus:

- Onion: Very domain-driven. It's all about keeping the domain model central.
- Clean: Focuses more on framework independence and flexibility with use cases driving the logic.

Use Cases and Adaptability:

- Onion: Focuses on the domain and business rules.
- Clean: Introduces use cases or interactors to handle application-specific rules, making it adaptable to changing system needs.

Testability and Maintainability:

- Both architectures support unit testing, but Clean Architecture is generally easier to adapt to new frameworks or system changes due to its flexibility.

Key Factors to Consider:

- **System complexity:** Do you need strict domain-driven design (Onion) or more flexibility (Clean)?
- **Framework dependencies:** Are you likely to change frameworks in the future? If so, Clean Architecture may be a better fit.
- **Long-term maintainability:** Both architectures help with maintainability, but Clean Architecture might offer better adaptability over time.

When to Use:

- **Onion Architecture**
  - **Domain-driven enterprise systems** with complex business logic.
    Why Use It? Strict separation of layers ensures that your domain logic remains isolated, making it easy to change infrastructure without impacting the core.
    Examples:
    - Banking, healthcare, or logistics applications where the domain model is central.
    - A healthcare system where the domain model (e.g., patient management) is the central part of the application.
- **Clean Architecture**
  - **Cloud-native applications and microservices** requiring framework independence and flexibility.
    Why Use It? Clean Architecture's flexibility makes it ideal for projects where you need the ability to swap frameworks or adapt quickly without affecting your core logic.
    Examples:
    - E-commerce platforms, SaaS applications, or systems that need to adapt quickly to evolving frameworks and technologies.
    - A microservice in an e-commerce platform where the system must adapt to different technologies like databases, message queues, or front-end frameworks.

پیوست ۲ : اصول طراحی Design Principles

Design Principles

Goal of Design Principles
Code must be KISS, DRY and YAGNI!

1) Design principles[1] should lead to good object-oriented design.
2) Code and architecture smells are based on the violation of accepted design principles.
3) Provide important information on how to fix code and architecture smells.
4) If the violated design principle can be identified, it provides a first indication of what a better structure for the system might look like.
5) Have been published and propagated by Robert C. M

SoC (Separation of concerns[293]) (single responsibility architecture)

- Keep related things together, unrelated things apart.
- A component should have exactly one task.
- Avoid mixing of several responsibilities in e.g. one class.
- Components become simpler, easier to understand, maintain and have better reusability.
- The complex of tasks of a unit should be self-contained (high cohesion).
- The unit should depend as little as possible on other units (low coupling).
    - N-tier architecture

DRY (Don't repeat yourself)

- Refers to the avoidance of duplicated code, i.e. code fragments that are implemented in the same or very similar way in several places.
- Redundant existing source code is difficult to maintain, as consistency between the individual duplicates must be ensured.
- In systems that remain loyal to the DRY principle, however, changes need only be made in one place.

SOLID[294] (object-oriented design)

1) **Single responsibility principle (SRP):** Every class should have only one responsibility.
2) **Open-closed principle (OCP):** This rule is about an Inheritance and Abstraction. Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
3) **Liskov substitution principle (LSP):** Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
4) **Interface segregation principle (ISP):** Clients should not be forced to depend upon interfaces that they do not use.
5) **Dependency inversion principle (DIP):** Depend upon abstractions, [not] concretions.

KISS (Keep it simple, stupid)

- Think of several options and then try the simplest option first.
- Always ask: Is there an easier way to do this?
- Think about the future maintainers, assume that they will be you, and work on that basis.
- When you look at existing systems, ask: Do I need all this stuff or Do we need it at all?
- Means that always the simplest solution of a problem should be chosen.
- Avoid code that is too complex and therefore too complicated.
- Finding simple solutions is a rule that helps to avoid various errors.
- A healthy rejection of non-simple solutions is a sign of searching for good code.
- The more difficult code is to explain, the more likely it is that it is more complicated than necessary and is not the most elegant solution.
- Make things as simple as possible, but not simpler!

YAGNI (You Aren't Gonna Need It)

- The architecture should be intended to support current and future requirements agreed with business stakeholders.
- Where no requirement, also no implementation!
- A program should only implement functionality when this functionality is needed.
- Contrary to this approach, in practice it is often attempted to prepare programs for possible future change requests and features through additional or more general code. "…this is what THEY will demand soon anyway…"
- Such code can be very annoying to read, understand, change existing functionality and costs time and money!

293 - https://en.wikipedia.org/wiki/Separation_of_concerns
294 - https://en.wikipedia.org/wiki/SOLID , https://www.tutorialsteacher.com/csharp/solid-principles

POLA (Principle of least astonishment)

- Keep things consistent so that people are not surprised when they find that a similar task is being done in a different place in a different way.
- If a difference is needed, document why.
- Consistency guides understanding, if you name things wrong or call the same thing by different names, you increase complexity.

LoD (The Law of Demeter) (a less well-known principle) (principle of least knowledge)

1) A unit should have only limited knowledge about other units: only units "closely" related to the current unit.
2) A unit should only talk to its immediate friends
3) A unit should not talk to strangers

SCP (Speaking Code Principle) Writing code that speaks for itself and that does not need a comment.

- Code should communicate its purpose, even without comment and documentation.
- Comments are no substitute for bad code.
- Clear and expressive code with few comments is much better than messy and complex code with numerous comments.
- Instead of spending time writing comments that explain the chaos, spend time on cleaning up the chaos.
- Comment must be adjusted when the code is changed.

Occam's Razor principle

Occam's Razor states the following: *Entities should not be multiplied without necessity*. To paraphrase, this essentially means that *the simplest solution is most likely the correct one*. So, in software development, the breaking of the principle of Occam's Razor is accomplished by making unnecessary assumptions and employing the least simple solution to a software problem.

Software projects are usually founded upon a collection of **facts** and **assumptions**. Facts are easy to deal with but assumptions are something else. When coming up with a software project solution to a problem, you normally discuss the problem and potential solutions as a team. When choosing a solution, you should always choose the project with the least assumptions as this will be the most accurate choice to implement. If there are a few fair assumptions, the more assumptions you are having to make, the more likely it is that your design solution is flawed.

A project with less moving parts has less that can go wrong with it. So, by keeping projects small with as few entities as possible by not making assumptions unless they are necessary, and only dealing with facts, you adhere to the principle of Occam's Razor.

OOD[295]

Four Pillars of OOPs (object-oriented programming) in Your Projects

1) Data abstraction is the process of hiding unnecessary details of an object's internal structure. By abstracting an object's data, its structure and behavior can be kept separate and more easily understood.
2) Encapsulation is the process of wrapping data and related functions into a single unit (object). Encapsulation limits access to object data and methods, preventing their misuse and ensuring their proper functioning.
3) Inheritance is the ability to create a new class (child class) from an existing one (parent class). The child class typically inherits the attributes (members and methods) of the parent class, although it can also redefine them.
4) Polymorphism is the ability of an object to take on multiple forms. This allows objects of different classes to be used interchangeably, as long as they implement a certain interface (have methods of the same name).

- Encapsulation is a way to keep code organized and separate from other codes in the system. It is done by using classes that contain all the code related to a specific feature or subject.
- Abstraction is the process of hiding internal details or processes from the user. This is done by creating classes that represent real-world objects and their attributes.
- Inheritance is when a class inherits the methods and properties of another class. This allows developers to reuse code and easily add new features.
- Polymorphism allows developers to create multiple methods with the same name, but with different implementations. This makes it easier to create flexible systems that are easier to understand and maintain.

Finally, the fifth principle is modularity, which means that a program should be broken down into smaller, more manageable pieces. This makes it easier to update and maintain the code and reduces the risk of introducing bugs.

[295] - https://en.wikipedia.org/wiki/Object-oriented_analysis_and_design , https://en.wikipedia.org/wiki/GRASP_(object-oriented_design) , https://en.wikipedia.org/wiki/Object-oriented_modeling

پیوست ۳ : بهروشها(روال مطلوب) Best Practices

Information security (CIA triad)

1) Confidentiality
2) Integrity (یکپارچگی (در ساخت) استحکام، انسجام)
3) Availability

CAP & PACELC theorem (Distributed Sytems)

1) **Consistency:** Every read receives the most recent write or an error. (ثبات یکپارچگی، انسجام)(هماهنگی، سازگاری)
2) **Availability:** Every request receives a (non-error) response, without the guarantee that it contains the most recent write or data.
3) **Partition tolerance:** The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes. Partition tolerance refers to the tolerance of a storage system to failure of a network partition. Even if some of the messages are dropped or delayed the system continues to operate.

Onion Architecture

1) Core Domain: (Should be isolatd from the outside world)
    a. Entity
    b. Commands (Actions)/Events (Result of actions)/Messages (Commands & Queries)
    c. Domain Services
2) Non-Core Domain:
    a. Database/Repositories
    b. File systems
    c. 3rd parties' systems
3) Non-Domain:
    a. Application Services
    b. UI

Mindsets

1) Reactive Manifesto https://www.reactivemanifesto.org/
2) Frameworkless Manifesto https://www.frameworklessmovement.org/ https://github.com/frameworkless-movement/manifesto
3) spotify squad/guild model https://www.atlassian.com/agile/agile-at-scale/spotify
4) Manifesto for Agile Software Development
5) Twelve Principles of Agile Software https://agilemanifesto.org/principles.html

Architectures

1) Microservices architecture
    a. Saga Orchestration vs Choreography vs Hybrid approach
2) Stateless services architecture
3) Event-driven architecture
4) Data storage and access architecture
5) Resiliency architecture
6) Evolution and operations architecture

Domain-Driven Design 4-tier architecture

1) Presentation Layer
2) Domain Layer (Business Part)(No Dependency)
3) Infrastructure Layer (Depend on Domain Layer)
4) Application Layer (Technical Part)(Depend on Domain Layer & Infrastructure Layer)

Anaemic Domain Model vs. Rich Domain Model

Facade pattern[296]

a software-design pattern commonly used in object-oriented programming

[296] - https://en.wikipedia.org/wiki/Facade_pattern

Other Robustness[297] principles

1) CQRS[298] and Event Sourcing "Command and Query Responsibility Segregation"
    a. SQL/Relational: High normal forms are good for commands(Create, Update, Delete)
    b. NoSQL : Low normal forms are good for queries(Read/Select)
    c. -- Note : CAP & PACELC theorem
2) GRASP[299] (object-oriented design) "General Responsibility Assignment Software Patterns (or Principles)"
3) DTSTTCPW "do the simplest thing that could possibly work"
4) GraphQL

SaaS (twelve-factor[300] app)

In the modern era, software is commonly delivered as a service: called web apps, or software-as-a-service. The twelve-factor app is a methodology for building software-as-a-service apps that:

1) Use declarative formats for setup automation, to minimize time and cost for new developers joining the project;
2) Have a clean contract with the underlying operating system, offering maximum portability between execution environments;
3) Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration;
4) Minimize divergence between development and production, enabling continuous deployment for maximum agility;
5) And can scale up without significant changes to tooling, architecture, or development practices.

The Twelve Factors

```
1)  Codebase            : One codebase tracked in revision control, many deploys
2)  Dependencies        : Explicitly declare and isolate dependencies
3)  Config              : Store config in the environment
4)  Backing services    : Treat backing services as attached resources
5)  Build, release, run : Strictly separate build and run stages
6)  Processes           : Execute the app as one or more stateless processes
7)  Port binding        : Export services via port binding
8)  Concurrency         : Scale out via the process model
9)  Disposability       : Maximize robustness with fast startup and graceful shutdown
10) Dev/prod parity     : Keep development, staging, and production as similar as possible
11) Logs                : Treat logs as event streams
12) Admin processes     : Run admin/management tasks as one-off processes
```

Design patterns[301]

Gang of Four (GoF (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) **23** Classic Design Patterns 1994)

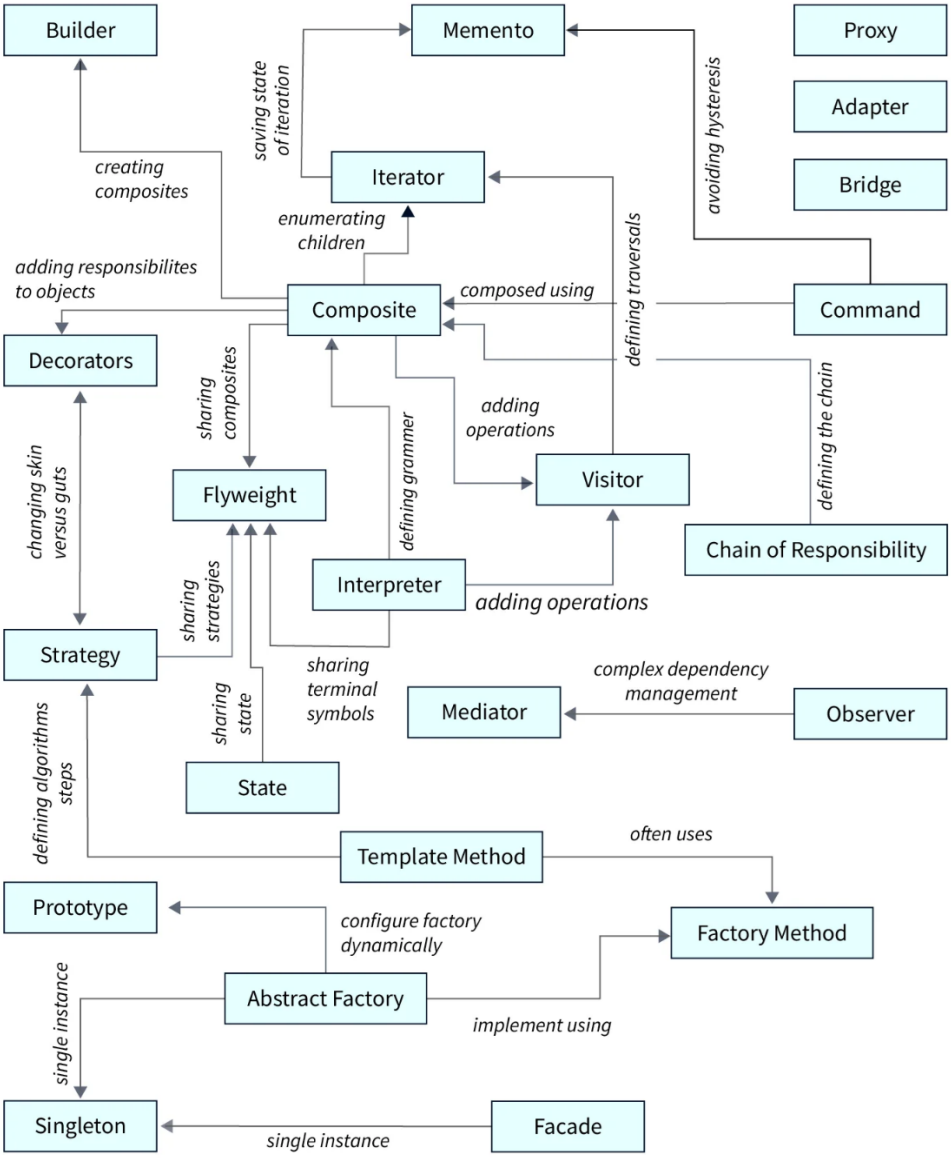
A. Creational Patterns: Focus on object creation mechanisms, aiming to create objects in a manner suitable for the situation.
1) Abstract factory groups object factories that have a common theme.
2) Builder constructs complex objects by separating construction and representation.
3) **Factory method** creates objects without specifying the exact class to create.
4) Prototype creates objects by cloning an existing object.
5) **Singleton** restricts object creation for a class to only one instance.

B. Structural Patterns: Deal with the composition of classes and objects, forming larger structures while maintaining flexibility.
1) **Adapter** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
2) Bridge decouples an abstraction from its implementation so that the two can vary independently.
3) Composite composes zero-or-more similar objects so that they can be manipulated as one object.
4) Decorator dynamically adds/overrides behavior in an existing method of an object.
5) Facade provides a simplified interface to a large body of code.
6) Flyweight reduces the cost of creating and manipulating a large number of similar objects.
7) Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

C. Behavioral Patterns: Address communication and interaction between objects, defining how objects collaborate to achieve a task. Examples include **Chain-of-Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor.**
1) Chain of responsibility delegates commands to a chain of processing objects.
2) Command creates objects that encapsulate actions and parameters.

297 - https://en.wikipedia.org/wiki/Robustness_principle
298 - https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
299 - https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)
300 - https://12factor.net
301 - https://en.wikipedia.org/wiki/Software_design_pattern

3) Interpreter implements a specialized language.
4) Iterator accesses the elements of an object sequentially without exposing its underlying representation.
5) Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
6) Memento provides the ability to restore an object to its previous state (undo).
7) **Observer** is a publish/subscribe pattern, which allows a number of observer objects to see an event.
8) State allows an object to alter its behavior when its internal state changes.
9) **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime.
10) Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
11) Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.



Anti patterns[302]

1) Spaghetti Code
2) Golden Hammer
3) Boat Anchor
4) Dead Code
5) God Object and God Class
6) Copy and Paste Programming
7) Nested Callback Hell Functions (in javascript)

302 - https://www.scaler.com/topics/software-engineering/anti-patterns/

Avoiding Software Anti-Patterns with Better System Management
- Perform Frequent Code Reviews
- Engage in Code Refactoring
- Make it Visual

Clean Code Fundamentals

1) Modular
2) Understandable (***Clear is better than Clever***)
3) Reliable
    a. Correctness
    b. Stable
    c. Resilience
4) Readable
5) Testable
6) Learnable
7) Configurable
8) Changeability
9) Extensibe
10) Efficiency
    a. Time
    b. Space
    c. Effects
11) Maintainability
    a. Adaptabillity
    b. Familiarity
12) Reusable:
    a. **modularity** (The functionality of your program should be divided into independent modules, each of which contains a part of the solution. Changes in a module or function shouldn't affect the rest of the code.)
    b. **high cohesion** (all the pieces in a module belong together)
    c. **low coupling** (modules are independent of each other)
    d. **separation of concerns** (the parts of a program should overlap in functionality as little as possible)
    e. **information hiding** (internal changes in a module shouldn't affect the rest of the system)

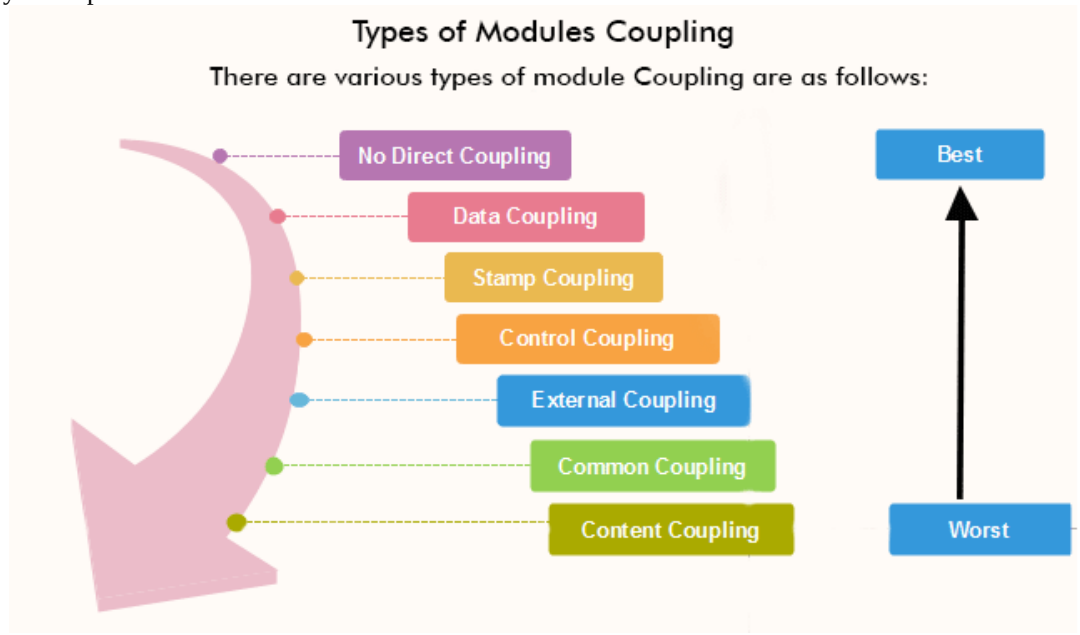
Design is good when

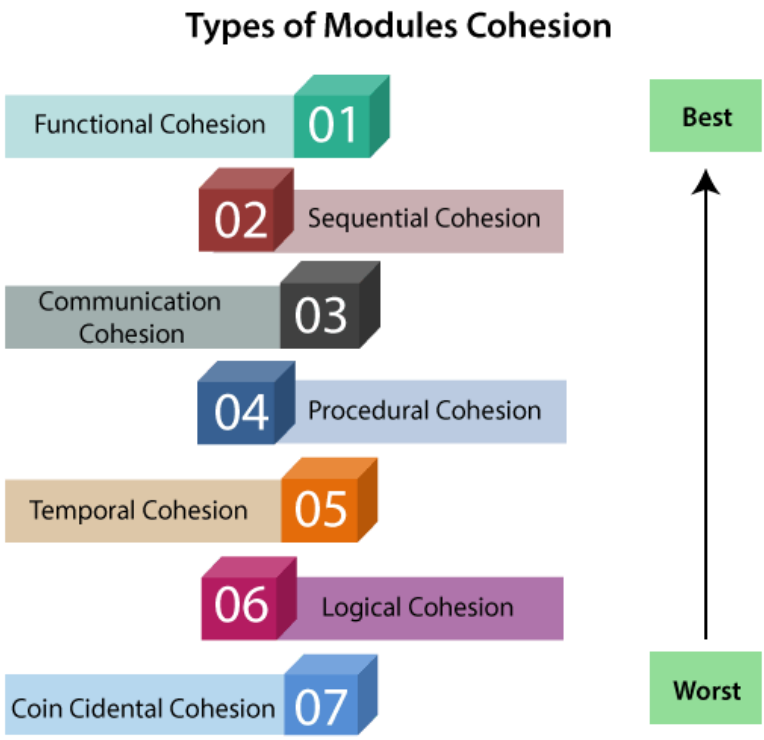
- it breaks down the complexity of the software into manageable and simple problems.
- small interfaces were defined.
- the components are decoupled.
- the components have clearly defined responsibilities.
- the software is maintainable.
- the software can be easily changed and extended.
- the software is stable.
- bugs can be fixed quickly.
- the software is reusable in other software projects.
- the code is understandable.

Criteria for good design

1) Correctness
    a. Fulfilment of requirements
    b. Playback of all functions of the system model
    c. Ensuring the non-functional requirements
2) Comprehensibility
    a. Self-explanatory code and design
    b. Good documentation

3) **High cohesion**[303] (Intra-Module Binding: relationship within the module.)
4) **Low coupling (**Inter-Module Binding: relationships between modules)
5) Reusability
6) Customizability
7) No cyclic dependencies





Symptoms of bad design

1) Never-touch-running-code Syndrome
    a. Developers are afraid to change code.
    b. Many workarounds, code is developed around it.
    c. Changes have unknown and undetected side effects.
2) Small change in requirements leads to big changes in code

[303] - https://www.javatpoint.com/software-engineering-coupling-and-cohesion

3) Reuse through code duplication (Copy-Paste)
    a. Developers chase after the places that need to be changed.
    b. The more code the more difficult it becomes to keep track of duplicates.
    c. Errors have to be patched several times at different places.
4) Cyclic relations between artifacts
    a. Artifacts that are cyclically coupled cannot be tested individually.
    b. Artifacts that are used in different cycles often play several roles, which makes them difficult to understand.
    c. Artifacts that are used in different cycles cannot be exchanged easily.

Concurrency vs. Parallelism

Concurrency:

The recognition that we can divide up a computation (an algorithm) into separate pieces where the order of execution of the pieces doesn't matter. Concurrency is a function of the algorithm you use to solve a problem. This also means that even if you create a concurrent program, if you run it on a single processor, it will still run serially.

Parallelism:

The mechanism used to execute a program on a particular machine or machine architecture in order to improve the performance of the program. Parallelism allows a concurrent program to execute on many different processors and thus potentially improve the overall performance of the program.

Features:

- **OS-native Threads:** when we need to manage and create concurrent or parallel execution of tasks.
- **Event-driven (Event loop):** This model is often used in combination with callbacks. It performs very well in terms of performance, but its biggest problem is the risk of callback hell.
- **Async/Await model:**
    - I/O operations
    - Web Development
    - Concurrency and parallelism
- **Message Passing (Channels/Buffers (CSP-based)):**
    - Inter-thread communication
    - Synchronization
    - Data Sharing
- **Actor Model:**
    - Concurrency and parallelism
    - Distributed system
    - Iot (Internte of Things)
- Mutex (Rust):

Monolithic architecture challenges

1) Large code base: This is a scenario where the code lines outnumber the comments by a great margin. As components are interconnected, we will have to bear with a repetitive code base.
2) Too many business modules: This is in regard to modules within the same system.
3) Codebase complexity: This results in a higher chance of code-breaking due to the fix required in other modules or services.
4) Complex code deployment: You may come across minor changes that would require whole system deployment.
5) One module failure affecting the whole system: This is in regard to modules that depend on each other.
6) Scalability: This is required for the entire system and not just the modules in it.
7) Intermodule dependency: This is due to tight coupling.
8) Spiraling development time: This is due to code complexity and interdependency.
9) Inability to easily adapt to a new technology: In this case, the entire system would need to be upgraded.

Recommendations

The app needs multiple instances to handle load.
The app needs SSL/TLS termination (for Network Load Balancers).
The app needs to redirect HTTP requests to HTTPS.
The app needs to serve multiple domains.
The app needs to serve static resources, e.g. jpeg files.

1) Use A Reverse Proxy
    a. HAProxy
    b. Nginx
2) Kubernetes (Auto Scaling)
3) Capacity Planning for Production

4) Running Multiple Instances

Programming paradigms[304]

1) Action
2) Array-oriented[305]
3) Automata-based
4) **Concurrent computing**
    a. **Actor-based**
    b. Choreographic programming
    c. Multitier programming
    d. Relativistic programming
    e. Structured concurrency
5) Data-driven[306]
6) Data-oriented[307]
7) **Declarative (contrast: Imperative)**
    a. **Functional**
        i. **Functional logic**
        ii. **Purely functional**
    b. Logic
        i. Abductive logic
        ii. Answer set
        iii. Concurrent logic
        iv. Functional logic
        v. Inductive logic
    c. Constraint
        i. Constraint logic
            1. Concurrent constraint logic
    d. Dataflow
        i. Flow-based
        ii. Reactive
            1. Functional reactive
    e. Ontology
    f. Query language
8) Differentiable
9) Dynamic/scripting
10) Event-driven
11) Function-level (contrast: Value-level)
    a. Point-free style
        i. Concatenative
12) Generic
13) **Imperative (contrast: Declarative)**
    a. **Procedural**
    b. **Object-oriented**
14) Intentional
15) Language-oriented[308]
    a. Domain-specific
16) Literate
17) Macroprogramming
18) Metaprogramming
    a. Automatic
        i. Inductive programming
        ii. Program synthesis
    b. Reflective
        i. Attribute-oriented[309]

304 - https://en.wikipedia.org/wiki/Programming_paradigm
305 - https://en.wikipedia.org/wiki/Array_programming
306 - https://en.wikipedia.org/wiki/Data-driven_programming
307 - https://en.wikipedia.org/wiki/Data-oriented_design
308 - https://en.wikipedia.org/wiki/Language-oriented_programming
309 - https://en.wikipedia.org/wiki/Attribute-oriented_programming

    c. Macro
    d. Template
19) Natural-language programming
20) Non-structured (contrast: Structured)
    a. Array
21) Nondeterministic
22) Parallel computing
    a. Process-oriented[310]
23) Probabilistic
24) Quantum
25) Set-theoretic
26) Stack-based
27) Structured (contrast: Non-structured)
    a. Block-structured
    b. Object-oriented[311]
        i. **Agent-oriented[312]**
        ii. Class-based
        iii. Concurrent
        iv. Prototype-based
        v. By separation of concerns:
            1. **Aspect-oriented[313]**
            2. **Role-oriented[314]**
            3. **Subject-oriented[315]**
    c. Recursive
28) Symbolic
29) Value-level (contrast: Function-level)

[310] - https://en.wikipedia.org/wiki/Process-oriented_programming
[311] - https://en.wikipedia.org/wiki/Object-oriented_programming
[312] - https://en.wikipedia.org/wiki/Agent-oriented_programming
[313] - https://en.wikipedia.org/wiki/Aspect-oriented_programming
[314] - https://en.wikipedia.org/wiki/Role-oriented_programming
[315] - https://en.wikipedia.org/wiki/Subject-oriented_programming

پیوست ۴ : فناوریها/تکنولوژیها

Criteria for Evaluating Technology/Stack/API/Frameworks/Tools/Engines/Libs/…

There is no framework that is better than all, they all have their pros and cons. It all depends a lot on the requirement, the domain of the problem or the platform to be developed. Another false premise that we usually find is that we are inclined to choose a framework because of its popularity when popular does not necessarily mean better, it all depends on the need. Every framework has its trade-off, absolutely all have their points for and against.

Required criteria (Dominant):

- Maturity, Ecosystem, Built-in integrations and Community Support (Active community of users, Regular Updates and Long-term Viability)
- Scalability, Stability, Resiliency, Reusability, Observability, Durability, Performance[316] (TPS/RPS) and features (Load Times, Efficiency in Rendering (SSR, CSR, SSG, ISR, SPA), Optimization Ability)
- Must fit the project requirements (Fits your tech stack nicely. avoid "When you have a hammer, every problem looks like a nail")
- Suitable Use cases or senarios that can implement by candidate Technology.
- Popularity: How many Companies/Projects/Programmers around us, use this framework **(Network Effect)**[317]
- Future Proof: Compatibility in future releases (The Technology we choose is a **long-term commitment**.)
- Fits your team's Skills/Experience and Availability of Talent
- Cost of Maintenance and Development (Ability to hire developers and Aligns with your budget)
- Supports your time-to-market goals
- Project Requirements, Complexity and Architecture
    - Small Projects or MVPs
    - Large-scale or Enterprise Applications

Features (important but probably not a deal-breaker):

- Use Modern Programming Language Preference **(Typescript, C#, Rust, Go, Scala, Carbon, Gleam, …)**
- Parallelism Model: (Shared Memory, Message passing (Asynchronous/Synchronous))
    - **Event Loop** (JavaScript)
    - **Multi-threading** (C++, Java, C#)
    - **CSP-based** (Occam, Crystal, Go)
    - **Actor model** (Erlang/Elixir/LFE/Gleam, Scala/Akka, D, Dart, Ruby, Haskell, Swift, Rust, …)
- Architecture Patterns and Use Cases
- Legacy/Modern Architecture and Design of Framework (Asynchronous programming, non-blocking I/O, Reactive, Virtual Threads, user-space concurrency mechanisms, ...)
- Support Modern Technologies (Microservice, Container, Kubernetes, Serverless(Pay-per-use pricing and Automatic scaling)[318], Cloud computing service integration, Horizontal scaling, ...)
- Cloud computing services & solution (Microsoft Azure, Google Cloud Platform, Amazon AWS)
- Robust ORM (Object-Relational Mapping)
- Support for Modern Development Practices (CI/CD code pipeline, DevOps, and Agile methodologies)
- Integration with Existing Technologies (Easy to integrate with modern tools and cloud services, etc GitHub, Gitlab, Jira, Jenkins, Teams, Docker, Azure Devops, …)
- Dependency injection, modularization, and configuration management for maintainable, testable, scalable architecture.
- Abstractions and utilities for building RESTful APIs, GraphQL endpoints, or real-time communication using WebSocket.
- Middleware layers to process data, handle errors, manage sessions, or validate requests.

[316] - https://web-frameworks-benchmark.netlify.app/result , https://www.techempower.com/benchmarks/#hw=ph&test=db&section=data-r23
[317] - https://en.wikipedia.org/wiki/Network_effect , https://en.wikipedia.org/wiki/Reed_law
[318] - Event-driven architectures and API-first services: Serverless Framework (multi-language), SST (for AWS Lambda), Netlify Functions (JavaScript/Go), Firebase Functions (JavaScript).

- Extensive Libraries and Tools
- Cross platform and Supported Operating Systems
- Code compile to native image (for example **Java + GraalVM**)

Optional (nice to have):

- Universal Ecosystem (Platform Extensive) (Web, Desktop, Mobile, Game, IoT, AI, Cloud, ...)
- Open source & Open-source Organizations/Foundations Support[319]
- Number of Contributors/Forks/Stars on github
- Good documentation, Tutorials and Example code (that compiles and runs!)
- Learning Curve
- Available Project Templates[320]
- Who Support the Framework Project (Google, Microsoft, Redhat, Oracle, Meta, ...)
- Trends & Future of Technology
- Works without needing kludges or hacks
- Pricing, Fees
- Vendor's and 3rd party support
- IDEs and DevTools support
- Compatible license

Red Flags (Warning Signs):

- Declining GitHub activity
- Outdated documentation
- Poor security track record
- Limited community support
- Unclear upgrade paths

FUNDAMENTALS

1) Custom providers
2) Asynchronous providers
3) Dynamic modules
4) Injection scopes
5) Circular dependency
6) Module reference
7) Lazy-loading modules
8) Execution context
9) Lifecycle events
10) Platform agnosticism
11) Testing

TECHNIQUES

1) Configuration
2) Database
3) Mongo
4) Validation
5) Caching
6) Data Serialization Standards
    a. JSON/BSON (Binary JSON)
    b. XML
    c. TOML
    d. YAML (YAML Ain't Markup Language)
    e. CSON

[319] - https://en.wikipedia.org/wiki/List_of_free_and_open-source_software_organizations , https://www.apache.org/ , https://www.eclipse.org/ , https://www.fsf.org/ , https://www.cncf.io/ , https://www.oasis-open.org/ , https://www.osadl.org/ , https://opensource.org/ , http://gnu.org/

[320] - https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-new-sdk-templates
https://github.com/dotnet/templating/wiki/Available-templates-for-dotnet-new

    f. HCL
    g. HDF5(Hierarchical Data Format version 5)
    h. Ini
    i. CSV
    j. CBOR (Concise Binary Object Representation)
    k. Protobuf/gRPC
    l. Kryo
    m. Apache Avro
    n. Apache Thrift
7) Versioning
8) Task scheduling & Cron jobs
9) Queues
10) Logging
11) Cookies
12) Events
13) Compression
14) File upload
15) Streaming files
16) HTTP module
17) Session
18) Model-View-Controller
19) Performance
20) Server-Sent Events
21) Sharding

SECURITY

1) Authentication (Who are you?)
2) Authorization (What are you allowed to do?)
3) Encryption and Hashing
4) Helmet
5) CORS
6) CSRF Protection
7) Rate limiting

GRAPHQL

1) Resolvers
2) Mutations
3) Subscriptions
4) Scalars
5) Directives
6) Interfaces
7) Unions and Enums
8) Field middleware
9) Mapped types
10) Plugins
11) Complexity
12) Extensions
13) CLI Plugin
14) Generating SDL
15) Sharing models
16) Federation
17) Migration guide

WEBSOCKET

1) Gateways
2) Exception filters
3) Pipes
4) Guards
5) Interceptors
6) Adapters

OPENAPI[321]

1) Types and Parameters
2) Operations
3) Security
4) Mapped Types
5) Decorators
6) CLI Plugin

Tech Stacks

1)	LAMP stack	https://wordpress.org/
2)	MEAN stack	https://www.youtube.com/
3)	MERN stack	https://www.netflix.com/
4)	Java stack	https://www.tesla.com/
5)	.NET stack	https://stackoverflow.com/
6)	JAMstack	https://www.netlify.com/
7)	Ruby on Rails (RoR) stack	https://www.airbnb.com/
8)	Python stack	https://www.pixar.com/
9)	Serverless stack	https://www.figma.com/
10)	AI-First Stack	https://openai.com/

Other RECIPES

1) **Idempotency**
2) REPL
3) CRUD generator
4) Passport (auth)
5) Hot reload
6) MikroORM
7) TypeORM
8) Mongoose
9) Sequelize
10) Router module
11) Swagger
12) CQRS[322] (Command and Query Responsibility Segregation)
13) Compodoc
14) Prisma
15) Serve static
16) Commander
17) Async local storage
18) Automock
19) Auto-scaling: The ability of an application to automatically scale its resource usage in response to changes in demand.
20) Load balancers
21) **Service Mesh:** Implementing network communication between microservices using a service mesh, such as Istio or Linkerd, to provide features such as traffic management, security, and observability. A configurable infrastructure layer for microservices applications that makes communication between service instances flexible, fast, and reliable.
22) CI/CD code pipelines: A series of automated processes for continuously building, testing, and deploying code.
23) Serverless computing: Computing model in which the cloud provider manages the infrastructure and automatically allocates resources capacity needed to run applications.
24) Circuit Breaker: Implementing a circuit breaker pattern to prevent cascading failures and improve the resilience of microservices.
25) API Gateway: Implementing an API gateway to manage and secure access to microservices and provide a single-entry point for external consumers.
26) **Strangler Pattern:** Gradually replacing parts of a monolithic application with microservices over time to minimize disruption and risk.

Inter-Process Communication Approaches (IPC)[323]

- File

[321] - https://www.openapis.org/ , https://en.wikipedia.org/wiki/OpenAPI_Specification , https://swagger.io/specification/ , https://swagger.io/resources/open-api/ , https://orpc.unnoq.com/ , https://github.com/unnoq/orpc , https://trpc.io/ , https://github.com/trpc/trpc

[322] - https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs
https://martinfowler.com/bliki/CQRS.html

[323] - https://en.wikipedia.org/wiki/Inter-process_communication

- Signal[324]; also, Asynchronous System Trap[325]
- Socket[326]
- Message queue[327] (message-oriented middleware MOM[328])
- Anonymous pipe[329]
- Named pipe[330]
- Shared memory[331]
- Message passing[332]
- Memory-mapped file[333]
- Unix domain socket[334]
- Communicating sequential processes[335]

Deployment strategies

1) **Basic deployment**: is often the simplest form of deployment, involving a straightforward process where the software is installed and configured in a live environment. This approach may suit smaller applications or systems that do not require complex configurations or extensive testing environments. However, basic deployment comes with high risks, as there is little room for testing or rollback if something goes wrong.
2) **Big Bang Deployment:** A big bang deployment is a type of software deployment in which all of the changes are deployed to the production environment all at once. This causes a bit of downtime as we have to shut down the old system to switch on the new one. The downtime is usually short, but be careful, it can stay. If things don't go as planned, preparation and testing are key. If things go wrong, we roll back to the previous version. Rolling back is not always pain-free though we still may disrupt users and there could still be data implications. We need to have a solid rollback plan. Big Bang is sometimes the only choice. For example, when an intricate database upgrade is involved, then we have the rolling deployment.
3) **Continuous Deployment (CD):** is a software deployment strategy that allows you to release new versions of your application to production at any time, without human intervention.
4) **Recreate:** Version A is terminated then version B is rolled out.
5) **Rolling-update (also known as Ramped slow or Incremental):** Version B is slowly rolled out and replacing version A. It gradually replaces old pods with new ones, ensuring that some pods are always running during the update. Rolling deployment involves gradually replacing instances of the previous software version with the new one. This reduces downtime and allows for testing the new software in a live environment incrementally. If issues are detected, the deployment can be paused or rolled back, minimizing disruptions to the user experience.
   Rolling deployment is particularly useful for systems requiring **high availability** since it permits partial upgrades without taking the entire system offline. It allows for immediate feedback and continuous improvement, ensuring that any faults can be addressed quickly without impacting the overall service performance.
   Here's an example of how it might work. Imagine we have 10 servers running our application. In a rolling deployment, we might take down the first server, deploy the new version of our application there, and bring it back online. Once we've confirmed everything is working as expected, we'll move

[324] - https://en.wikipedia.org/wiki/Signal_(IPC)
[325] - https://en.wikipedia.org/wiki/Asynchronous_system_trap
[326] - https://en.wikipedia.org/wiki/Network_socket
[327] - https://en.wikipedia.org/wiki/Message_queue
[328] - https://en.wikipedia.org/wiki/Message-oriented_middleware
[329] - https://en.wikipedia.org/wiki/Anonymous_pipe
[330] - https://en.wikipedia.org/wiki/Named_pipe
[331] - https://en.wikipedia.org/wiki/Shared_memory
[332] - https://en.wikipedia.org/wiki/Message_passing
[333] - https://en.wikipedia.org/wiki/Memory-mapped_file
[334] - https://en.wikipedia.org/wiki/Unix_domain_socket
[335] - https://en.wikipedia.org/wiki/Communicating_sequential_processes

on to the second server, and so on. This approach allows the new version to gradually replace the old one, server by server until the entire system is updated.

6) **Blue/Green:** Blue/Green deployment is a zero-downtime strategy that uses two identical environments: Blue (live) and Green (staging/next). Version Green is released alongside version Blue, then the traffic is switched to version Green. Blue/green deployment uses two identical production environments. One environment (blue) runs the current version of the application, while the new software version is deployed to the other (green) environment. Traffic is switched to the green environment once the new version is confirmed to be stable and functional, ensuring a transition with minimal downtime. This method provides a **fail-safe mechanism**. If the new version encounters issues, traffic can be switched back to the blue environment, **ensuring service continuity**. Blue/green deployment significantly reduces risks and allows for testing in a production-simulated environment before fully switching over.
7) **Canary Releases:** Version B is released to a subset of users, then proceed to a full rollout. Canary deployment involves releasing the new version of software to a small subset of users before rolling it out to the entire user base. This gradual exposure allows for monitoring and gathering feedback on the new release while minimizing the risk of widespread issues. If the deployment proves successful, the rollout continues until all users are on the new version.
   You need strong monitoring in place for the canary approach, so you can quickly identify potential problems and halt the deployment if necessary. It works well in environments that demand **high availability,** enabling teams to validate changes and gather user feedback without impacting the entire user population.
8) **A/B testing:** Version B is released to a subset of users under specific condition. A/B testing can be used as part of your release process when you want to compare **different variations of a feature** and measure which one performs better. A/B testing is a method where **two versions of software** are deployed simultaneously to different user groups to compare qualities such as system performance or user response to a change. This approach helps identify the more effective version based on defined metrics such as user engagement, performance, or other key indicators.
   By analyzing the results from each group, teams can make data-driven decisions about which version should be fully deployed. This strategy ensures that the chosen version has been validated to provide the best outcome, leading to higher satisfaction and better overall performance.
9) **Feature Toggles (also known as Feature Flag):** are a powerful way to deploy code without activating it immediately. Think of them as remote-controlled switches for turning features on or off. This approach allows you to push code to production without activating new functionality until you're ready. This strategy supports gradual rollouts and A/B testing, ensuring that any risks associated with new features are minimized.
10) **Shadow Deployment / Dark Launching:** Version B receives real-world traffic alongside version A and doesn't impact the response.
11) **Rollback Strategy:** is a crucial aspect of deployment, allowing quick reverting to a previous version in case of issues.
12) **Multi-service deployment:** focuses on deploying multiple, interdependent services either simultaneously or in a sequence, depending on their relationships and dependencies. This method is commonly used in microservice architectures where different services perform specific functions within the application ecosystem.

Choosing the Right Strategy

There's no one-size-fits-all deployment strategy. Your choice depends on:

- Application criticality
- Size and complexity of the project
- Availability & scalability requirements
- Resources available
- Impact on end-users

- Downtime tolerance
- User base size and segmentation
- Infrastructure Capacity/Complexity
- Cost constraints
- Team Experties
- Need for rollback or testing(experimentation and analytics)

Strategy	ZERO DOWNTIME	REAL TRAFFIC TESTING	TARGETED USERS	CLOUD COST	ROLLBACK DURATION	NEGATIVE IMPACT ON USER	COMPLEXITY OF SETUP
RECREATE version A is terminated then version B is rolled out	✗	✗	✗	■□□	■■■	■■■	□□□
RAMPED version B is slowly rolled out and replacing version A	✓	✗	✗	■□□	■■■	■□□	■□□
BLUE/GREEN version B is released alongside version A, then the traffic is switched to version B	✓	✗	✗	■■■	□□□	■■□	■■□
CANARY version B is released to a subset of users, then proceed to a full rollout	✓	✓	✗	■□□	■□□	■□□	■■□
A/B TESTING version B is released to a subset of users under specific condition	✓	✓	✓	■□□	■□□	■□□	■■■
SHADOW version B receives real world traffic alongside version A and doesn't impact the response	✓	✓	✗	■■■	□□□	□□□	■■■

پیوست ۵ : انواع سوءگیرهای شناختی(موثر در شکست پروژه ها)

موارد مغالطه ها[۳۳۶] و سوءگیرهای شناختی[۳۳۷]:

۱) سوگیری احساسی Affect heuristic[۳۳۸]
۲) فرضِ عامل هوشمند Agent detection[۳۳۹]
۳) اثر ابهام Ambiguity effect[۳۴۰]
۴) لنگر انداختن Anchoring[۳۴۱]
۵) سوگیری توجه Attentional bias[۳۴۲]
۶) سوگیری خودکارسازی Automation bias[۳۴۳]
۷) خطای تمرکز و اعتماد بر اطلاعات در دسترس Availability Bias[۳۴۴]
۸) غفلت از نرخ پایه Base rate fallacy[۳۴۵]
۹) سوگیری اعتقادی (اعتقاد گرایی) Belief bias[۳۴۶]
۱۰) اثر ارابه موسیقی یا همرنگ شدن Bandwagon Effect[۳۴۷]
۱۱) سوگیری تأیید Confirmation bias[۳۴۸]
۱۲) انتخاب گزینشی یا گلچین کردن Cherry picking[۳۴۹]
۱۳) کوری انتخاب Choice Blindness
۱۴) سوگیری مهر طلب Courtesy bias[۳۵۰]
۱۵) اثر دانینگ–کروگر Dunning–Kruger effect[۳۵۱] : خطایی ادراکی که در آن اشخاص با کمی یادگیری در یک زمینه دچار توهم تخصص و خبرگی در آن زمینه می‌شوند.
۱۶) خطای توجیه تلاش Effort Justification[۳۵۲]
۱۷) اثر برخورداری Endowment effect[۳۵۳]
۱۸) خطای هزینه هدر رفته - تعهد بیشتر به خطا Sunk cost - Escalation of Commitment[۳۵۴]
۱۹) اثر قالب بندی یا تزئین کردن Framing effect
۲۰) سوگیری جمع‌آوری اطلاعات Information bias[۳۵۵]
۲۱) همبستگی پنداری Illusory correlation bias[۳۵۶]
۲۲) اثر حقیقت واهی Illusory truth effect[۳۵۷]

---

336 - https://en.wikipedia.org/wiki/List_of_fallacies
337 - https://en.wikipedia.org/wiki/List_of_cognitive_biases
338 - https://en.wikipedia.org/wiki/Affect_heuristic
339 - https://en.wikipedia.org/wiki/Agent_detection
340 - https://en.wikipedia.org/wiki/Ambiguity_effect
341 - https://en.wikipedia.org/wiki/Anchoring_effect
342 - https://en.wikipedia.org/wiki/Attentional_bias
343 - https://en.wikipedia.org/wiki/Automation_bias
344 - https://en.wikipedia.org/wiki/Availability_heuristic
345 - https://en.wikipedia.org/wiki/Base_rate_fallacy
346 - https://en.wikipedia.org/wiki/Belief_bias
347 - https://en.wikipedia.org/wiki/Bandwagon_effect
348 - https://en.wikipedia.org/wiki/Confirmation_bias
349 - https://en.wikipedia.org/wiki/Cherry_picking
350 - https://en.wikipedia.org/wiki/Response_bias#Courtesy_bias
351 - https://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect
352 - https://en.wikipedia.org/wiki/Effort_justification
353 - https://en.wikipedia.org/wiki/Endowment_effect
354 - https://en.wikipedia.org/wiki/Escalation_of_commitment
355 - https://en.wikipedia.org/wiki/Information_bias_(psychology)
356 - https://en.wikipedia.org/wiki/Illusory_correlation
357 - https://en.wikipedia.org/wiki/Illusory_truth_effect

۲۳) توهم اعتبار نظر Illusion of validity[۳۵۸]
۲۴) توهم کنترل Illusion of control[۳۵۹]
۲۵) تفکر وهمی یا خیالی Magical thinking[۳۶۰]
۲۶) اثر مواجهه صرف Mere-exposure effect[۳۶۱]
۲۷) استدلال هدفمند شده motivated reasoning[۳۶۲]
۲۸) سوگیری مایوپیا یا کوته بینی Myopia bias
۲۹) سوگیری داستان‌گویی (داستان‌سرایی) Narrative Bias
۳۰) ارتباط پنداری (آپوفینیا) Apophenia[۳۶۳]
۳۱) سوگیری انتظار مشاهده‌گر Observer-expectancy effect[۳۶۴]
۳۲) سوگیری آکام یا تصمیم بدیهی Occam's razor bias
۳۳) سوگیری اهمال (کاری نکردن) Omission bias[۳۶۵]
۳۴) سوگیری خوش‌بینی Optimism Bias[۳۶۶]
۳۵) اثر شترمرغ ostrich effect[۳۶۷]
۳۶) اثر بیش‌اعتمادی overconfidence effect[۳۶۸]
۳۷) سوگیری نتیجه‌نگر (نتیجه نگری) outcome bias[۳۶۹]
۳۸) اثر امکان یا خطای احتمال ممکن Possibility Effect
۳۹) سوگیری لجاجت یا مقاومت Reactance[۳۷۰]
۴۰) گذشته‌نگری مثبت Rosy retrospection[۳۷۱]
۴۱) دریافت گزینشی Selective perception[۳۷۲]
۴۲) اعطای مجوز کار نادرست به خود Self-licensing[۳۷۳]
۴۳) توهم تکرار Frequency illusion or Baader–Meinhof effect
۴۴) سوگیری انتخاب Selection bias[۳۷۴]
۴۵) اثر تأیید شخصی (اعتباردهی ذهنی) Subjective validation[۳۷۵]
۴۶) خطای جایگزینی Substitution bias[۳۷۶]
۴۷) خطای بقا یا سوگیری بازمانده Survivorship bias[۳۷۷]
۴۸) خطای کمبود Scarcity Bias[۳۷۸]

358 - https://en.wikipedia.org/wiki/Illusion_of_validity
359 - https://en.wikipedia.org/wiki/Illusion_of_control
360 - https://en.wikipedia.org/wiki/Magical_thinking
361 - https://en.wikipedia.org/wiki/Mere-exposure_effect
362 - https://en.wikipedia.org/wiki/Motivated_reasoning
363 - https://en.wikipedia.org/wiki/Apophenia
364 - https://en.wikipedia.org/wiki/Observer-expectancy_effect
365 - https://en.wikipedia.org/wiki/Omission_bias
366 - https://en.wikipedia.org/wiki/Optimism_bias
367 - https://en.wikipedia.org/wiki/Ostrich_effect
368 - https://en.wikipedia.org/wiki/Overconfidence_effect
369 - https://en.wikipedia.org/wiki/Outcome_bias
370 - https://en.wikipedia.org/wiki/Reactance_(psychology)
371 - https://en.wikipedia.org/wiki/Rosy_retrospection
372 - https://en.wikipedia.org/wiki/Selective_perception
373 - https://en.wikipedia.org/wiki/Self-licensing
374 - https://en.wikipedia.org/wiki/Selection_bias
375 - https://en.wikipedia.org/wiki/Subjective_validation
376 - https://en.wikipedia.org/wiki/Attribute_substitution
377 - https://en.wikipedia.org/wiki/Survivorship_bias
378 - https://en.wikipedia.org/wiki/Scarcity_(social_psychology)

۴۹) آرزو اندیشی Wishful thinking[۳۷۹]

۵۰) اثر شگفتی زنان Women are wonderful effect[۳۸۰]

۵۱) اصل پیتر[۳۸۱] : هر آنچه (درست) کار می‌کند در حالتی پیشرفته‌تر و چالش‌برانگیزتر به کار گرفته خواهد شد تا آنجا که دیگر کارایی از خود نشان ندهد.

۵۲) قانون پارکینسون[۳۸۲] : هر کار تا زمانی که برای آن تعیین شده طول می‌کشد و این مدت ارتباط چندانی با میزان و ماهیت کار ندارد.. و همچنین تمایل مدیران به داشتن مرئوسان بیشتر.

۵۳) قانون پیش‌پاافتادگی پارکینسون[۳۸۳] : اعضای یک سازمان به طرز نامتناسبی به مسائل پیش‌پاافتاده و جزئی اهمیت می‌دهند. در این مورد، زمانی‌که صرف هر موضوع مرتبط با طرح می‌شود، نسبت معکوس با میزان بودجه‌اش دارد.

۵۴) قانون ادوارد مورفی : هر خطای ممکن، حتما رخ خواهد داد. همه چیز ذاتاً دچار خطا و دردسر می‌شود مگر اینکه برای درستی آن تلاشی شده باشد.

۵۵) سو گیری ناشی از تجارب شخصی : توجه به مدل "چهار بت" فرانسیس بیکن (۱۶۲۶- ۱۵۶۱) فیلسوف و سیاستمدار برجسته بریتانیایی.

a. بت های غار idols of the Cave
b. بت های قبیله idols of the Tribe
c. بت های بازاری idols of Marketplace
d. بت های نمایشی idols of the Theater

379 - https://en.wikipedia.org/wiki/Wishful_thinking
380 - https://en.wikipedia.org/wiki/Women-are-wonderful_effect
381 - https://en.wikipedia.org/wiki/Peter_principle
382 - https://en.wikipedia.org/wiki/Parkinson%27s_law
383 - https://en.wikipedia.org/wiki/Law_of_triviality

پیوست ۶ : سازمان پروژه

وظایف کمیته مشاوره و نظارت فنی:

- **حکمرانی، هماهنگ سازی استراتژی ها، سیاستها، استانداردها، رویه ها، متریکها، بازخوردها، حل مسائل، ...**
- سیستم/فرآیند سازی، سیاستگذاری و نظارت فنی، تبیین راهبردها، راهکارها، معماریها، پلاتفرم ها، استاندارد ها، طراحی ها(مفهومی، منطقی، فیزیکی)، قالب ها، تعیین Best Practiceها، چک لیستها، رویه ها، دستورالعملها، آئین نامه ها، سناریوها، برنامه ریزی برای مقابله با چالشها و بحرانها، آسیب پذیریها، تهدیدها، ریسکها، شناسائی گلوگاه ها، ...
- تحقیق و توسعه تکنولوژیها R&D
- اندازه گیری تعالی/بلوغ/کارائی هر واحد، مدیریت و کنترل ریسک و احتمال وقوع و شکل/نحوه توزیع آنها، راستی آزمائی، تعیین میزان انحراف از اهداف و برنامه ها، چگونگی برونسپاری و واگذاری کارها، تعریف پروژه درست و چگونگی انجام درست آن و فرآیند تحویل گیری،RFP ، ...
- ترجمه محصولات و خدمات بالقوه(اعلام شده از لایه استراتژیک به لایه تاکتیکی) به نیاز مندیهای فنی و تکنولوژیکی بالفعل و مراقبت از کیفیت ارائه آن، Functional/Non-functional
- فرهنگ اجرا ، تمرکز بر ایجاد بستری که موفقیت را ممکن سازد، تعیین انتظارات روشن، کاهش سطح ابهام وظایف، جلوگیری از عملکرد ضعیف،
- اطمینان حاصل شود که اعضای تیم منابع، ابزار، پشتیبانی و اعتماد به نفس برای انجام مؤثر مسئولیت های شغلی خود را داشته، این شامل ارائه آموزش مداوم، دسترسی به اطلاعات و کمک به رفع موانع موفقیت است که مانع پیشرفت می‌شود
- مدیریت بحران، بازیابی از حملات/سوانح/حوادث، مدیریت SLA/OLA، Scheduled/Unscheduled Down Time ،
- مدیریت دانش بدست آمده (کمیت، کیفیت، چرائی، چگونگی، پایائی و روائی) ،
- KPI/OKR/Metrics، و همچنین تعیین سطوح سه گانه زیر و محدوده وظایف هر یک
  - Software Engineering : سطح مهندسی، معماری
  - Software Programming : سطح طراحی و برنامه سازی ، مستقل از زبان برنامه نویسی
  - Software Coding : سطح زبان برنامه نویسی
  - Engineering Productivity Metrics : اندازه‌گیری عملکرد تیم‌های توسعه با معیارهای کمی و کیفی به جای صرفاً زمان یا خطوط کد.

وظایف مدیران و کارشناسان:

- **معاونین C-Level :** ***تبیین و تعیین اهداف***، اطمینان از **ارائه مشاوره/آموزش و پیشنهادهای جذاب به مشتری**، مدیریت و کنترل پروژه ها، برنامه ریزی، تخصیص منابع، مدیریت و کنترل ریسک، نظارت بر حسن انجام کلیه موارد ابلاغ شده از سوی کمیته فنی بر زیر مجموعه خود، اصلاح و هموار سازی مسیر، نصب و عزل پرسنل، اطمینان از **صحت و دقت** در انجام امور و فرآیندها
- **مدیر :** ***سازماندهی پروژه ها***، اطمینان از **انجام عملیات در راستای رضایت و همراهی/همکاری با مشتری**، اولویت بندی، بررسی و تعیین وظایف و روشها، نقشها، پروسه ها، ارتباطات داخلی، مدیریت عملکرد، کنترل کمی/کیفی و کلی/ جزئی عوامل زیر مجموعه، اصلاح و تصویب گزارشات و مستندات، اطمینان از همگرا شدن برنامه ها/استراتژیها/اهداف و انتظارات تعیین شده با یکدیگر،
- **کارشناس مسئول :** ***مدیریت و کنترل فرآیندها***، اطمینان از **فراهم بودن و در دسترس بودن سرویس**، بررسی چالشها، راه حلها، زمان بندی، بهینه سازی راه کارها، اصلاح و تائید گزارشات و مستندات، اطمینان از همگرا شدن نتایج وظایف با اهداف تعیین شده، مدیریت/کنترل/هدایت تغییراتCR ها، Best Practice،
- **کارشناس :** اطمینان از **صحت و دقت** در ***انجام وظایف*** و فرآیندهای فنی/تکنیکی، تصحیح و بهینه سازی روشها ، تهیه گزارشات و مستندات مورد نیاز

پیوست ۷ : ATAM & CBAM

Architecture Tradeoff Analysis Method (ATAM)[384]

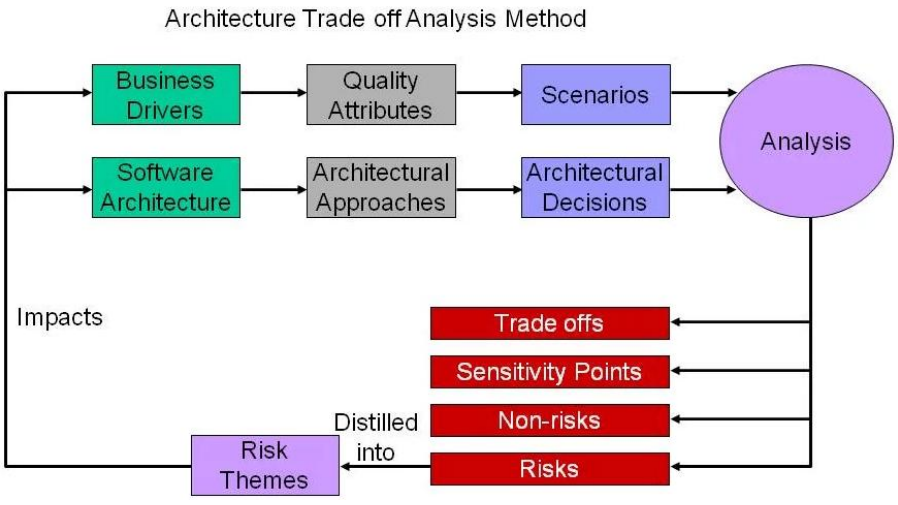
The Architecture Tradeoff Analysis Method (ATAM) is a structured technique for evaluating software architectures to determine how well they meet quality attribute goals and where potential risks exist. It involves gathering stakeholders to define business drivers and quality goals, creating scenarios to test architectural decisions, and analyzing the resulting trade-offs in a collaborative process. The goal is to identify architectural risks early in the development process so they can be addressed cost-effectively .

How ATAM works

- **Gather information:** An evaluation team meets with stakeholders (architects, users, developers, etc.) to understand business goals, system functionality, and desired non-functional properties like performance, security, and reliability.
- **Define scenarios:** The group develops "scenarios" based on the business drivers, which are specific, concrete situations that the architecture must handle (e.g., "How does the system handle a security attack?" or "How does it scale under heavy load?").
- **Analyze trade-offs:** The scenarios are used to analyze the architecture's design, focusing on how design decisions affect the quality attributes. This step identifies conflicts and potential risks where one quality goal might negatively impact another.
- **Report findings:** The analysis results in a concise presentation of the architecture, articulation of business goals, and a mapping of architectural decisions to quality requirements. It highlights risks and potential non-risks to inform decision-making .

Key outcomes

- **Identification of risks:** Early exposure of architectural risks that could inhibit the achievement of business goals.
- **Understanding of trade-offs:** Insight into how different quality attributes (like performance vs. security) interact and conflict with each other.
- **Informed decision-making:** A structured basis for negotiating inevitable trade-offs and making more informed design decisions.
- **Cost-effective solutions:** By identifying issues early, ATAM allows for problems to be solved when they are cheaper and easier to fix.



Who uses it

- **Software architects:** To evaluate their designs and ensure they meet business and technical requirements.
- **Stakeholders:** To provide input and understand how the architecture meets their specific needs.
- **Project decision-makers:** To gain clarity on the risks and benefits of a particular architectural approach.

ATAM benefits

[384] - https://en.wikipedia.org/wiki/Architecture_tradeoff_analysis_method , https://www.sei.cmu.edu/library/architecture-tradeoff-analysis-method-collection/ , https://www.geeksforgeeks.org/software-engineering/architecture-tradeoff-analysis-method-atam/ , https://www.sei.cmu.edu/documents/629/2000_005_001_13706.pdf , https://www.sei.cmu.edu/documents/2543/2018_010_001_515610.pdf

- identified risks early in the life cycle.
- increased communication among stakeholders.
- clarified quality attribute requirements.
- improved architecture documentation.
- documented basis for architectural decisions.
- Reduce Risk with Architecture Evaluation

Cost Benefit Analysis Method (CBAM)

CBAM is a process used to estimate the Return on Investment (ROI) of various software architectural design strategies. CBAM generally follows the Architectural Trade-off Analysis Method (ATAM) also developed by SEI. ATAM aids in defining scenarios and architectural strategies.

CBAM uses the output from ATAM as a starting point. It can be used independently from ATAM. CBAM helps to ensure that non-functional requirements or quality attributes are fully stated. CBAM elicits and documents the benefit (or utility) and cost associated with each architectural strategy defined. The cost of implementing each architectural strategy is also gathered from the stakeholders and then analyzed. The output from CBAM summarizes the ROI (Return On Investment) for each architectural strategy in consideration. ROI is calculated to show which strategy yields the most benefit. CBAM helps the developers choose the architecture design that will yield the best return on investment for the software system. There are few techniques found that incorporate the economic impact into software architecture analysis.

CBAM is an incremental nine-step process.

- First, stakeholders define, refine and further prioritize scenarios. The top 1/3 scenarios are selected based on priorities established.
- In Step 2 stakeholders define the response goals for the best, worst, current and desired cases for the selected scenarios.
- In Step 3 the stakeholders vote on each of the selected scenarios considering the expected response goals. 100 votes in total are allocated by the stakeholders. The votes are used to rank the priority of each scenario. The scenario with the most votes has the highest priority. Step 4 uses the top 50% of the scenarios from Step 3 based on votes (i.e. top 1/6 of the total).
- In Step 4, stakeholders assign a utility rating to each response goal (best, worst, current and desired) for the selected scenarios.
- In Step 5 stakeholders develop or review architectural strategies for the top 1/6 scenarios. A cost is assigned to each architectural strategy. The scenarios impacted by the strategy must also be defined.
- In Step 6, the stakeholders determine the expected response goals and utility rates for each scenario and architectural strategy.
- The benefit is calculated in Step 7. The benefit for each architectural strategy is based on the current utility rating and expected utility rating for each scenario impacted by the strategy. The benefit (b) for a given scenario (i) based on strategy (j) is calculated $b_{ij} = U_{expected} - U_{current}$. $U_{expected}$ is the utility expected if the scenario is implemented and $U_{current}$ is the current utility.
- In Step 8, the developer uses CBAM output to determine the architectural strategies that best suits the business needs.
- In Step 9, the developer must confirm results based on experience and intuition. For example, if a costly strategy that the developer expected to have lower benefit, is returned with the best ROI, the developer may need to review prior steps for accuracy and revisions may be required. The user can run an updated report and review again.

پیوست ۸ : عصر جدید مهندسی نرم افزار Agentic Software Engineering (agent-first era)[۳۸۵]

نگاه کنید به مقاله Agentic Software Engineering: Foundational Pillars and a Research Roadmap[۳۸۶] (Ahmed E. Hassan, 2025 Sep 27) و همچنین Spec-Driven Development[۳۸۷] (AI agents (AI-Driven Development Workflow for Your Team))

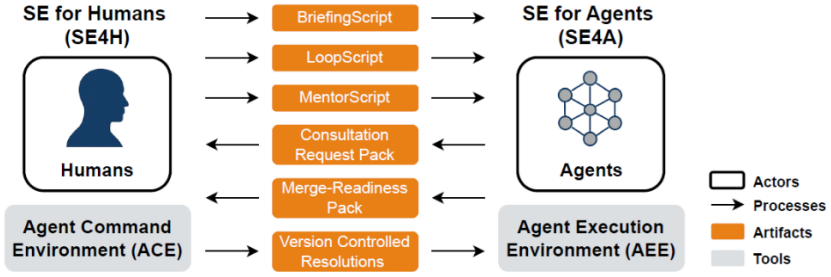


Fig. 1. Structured Agentic Software Engineering (SASE) overview.

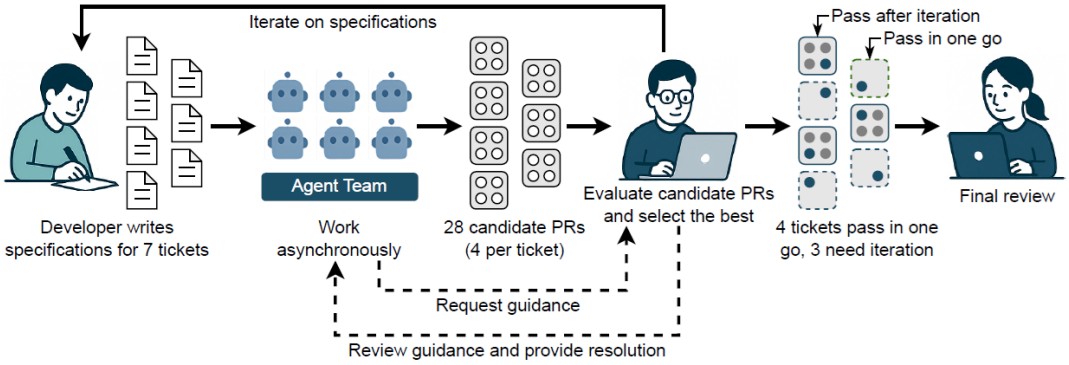


Fig. 2. Overview of an agentic SE workflow

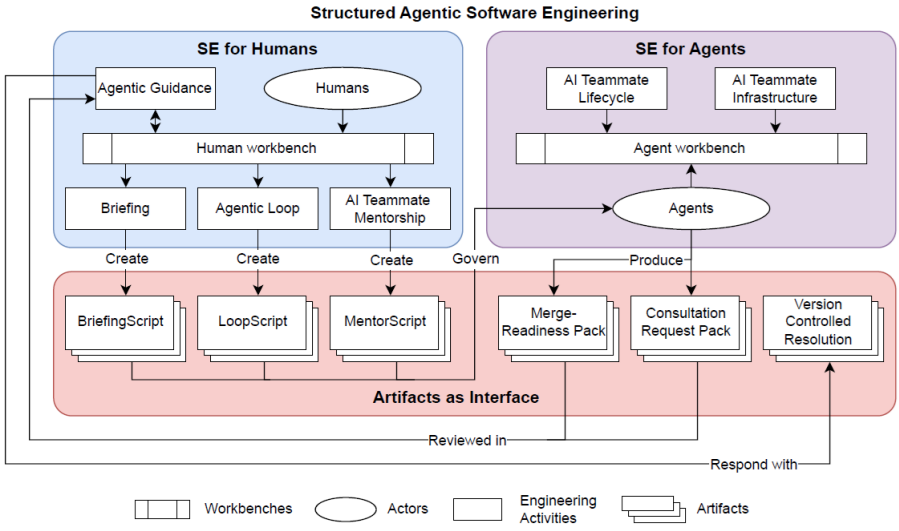


Fig. 3. The Structured Agentic Software Engineering (SASE) framework: dual domains for humans and agents, engineering activities, and artifacts.

[385] - https://azure.microsoft.com/en-us/blog/introducing-microsoft-agent-framework/ , https://github.com/microsoft/agent-framework https://learn.microsoft.com/en-us/agent-framework/overview/agent-framework-overview https://devblogs.microsoft.com/foundry/introducing-microsoft-agent-framework-the-open-source-engine-for-agentic-ai-apps/

[386] - https://arxiv.org/abs/2509.06216 , https://arxiv.org/html/2509.06216v1 , https://huggingface.co/papers/2509.06216

[387] - https://openspec.dev/ , https://github.com/github/spec-kit , https://kiro.dev/ , https://tessl.io/ , https://ainativedev.io/ , https://kilo.ai/ , https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html , https://github.com/github/spec-kit/blob/main/spec-driven.md, https://github.com/buildermethods/agent-os , https://developer.microsoft.com/blog/spec-driven-development-spec-kit

سایر ریفرنسها

http://everyspec.com/ANSI/
http://everyspec.com/NASA/
http://everyspec.com/NASA/NASA-NPR-PUBS/
http://everyspec.com/DoD/
http://everyspec.com/NATO/
http://everyspec.com/NIST/
https://nasa.github.io/progpy/npr7150.html
https://swehb.nasa.gov/display/SWEHBVD/C.+Project+Software+Requirements
https://nodis3.gsfc.nasa.gov/npg_img/N_PR_7150_002D_/N_PR_7150_002D_.pdf
https://ecma-international.org/

http://awesome-architecture.com/
http://awesome-architecture.com/clean-architecture/
http://awesome-architecture.com/clean-code/
http://awesome-architecture.com/data-driven-design/
http://awesome-architecture.com/domain-driven-design/domain-driven-design/
http://awesome-architecture.com/event-driven-architecture/
http://awesome-architecture.com/hexagonal-architecture/
http://awesome-architecture.com/onion-architecture/
http://awesome-architecture.com/rest/
http://awesome-architecture.com/scaling/
http://awesome-architecture.com/software-architecture/
http://principles-wiki.net/principles:start
https://deviq.com/antipatterns/code-smells
https://deviq.com/principles/principles-overview
https://github.com/RehanSaeed/.NET-Big-O-Algorithm-Complexity-Cheat-Sheet/tree/main
https://github.com/chanakaudaya/solution-architecture-patterns
https://java-design-patterns.com/principles/
https://deviq.com/antipatterns/code-smells
https://learn.microsoft.com/en-us/previous-versions/msp-n-p/hh917312(v=pandp.10)
https://www.simplethread.com/20-things-ive-learned-in-my-20-years-as-a-software-engineer/
https://github.com/sirius248/top-software-engineering-articles
https://www.geeksforgeeks.org/software-engineering/software-engineering-introduction-to-software-engineering/
https://www.geeksforgeeks.org/software-engineering/software-engineering/
https://softwareengineeringdaily.com/

https://en.wikipedia.org/wiki/Code_audit
https://en.wikipedia.org/wiki/Cost_estimation_in_software_engineering
https://en.wikipedia.org/wiki/Cyclomatic_complexity
https://en.wikipedia.org/wiki/DevOps
https://en.wikipedia.org/wiki/Extreme_programming
https://en.wikipedia.org/wiki/Halstead_complexity_measures
https://en.wikipedia.org/wiki/International_Software_Testing_Qualifications_Board
https://en.wikipedia.org/wiki/Programming_complexity
https://en.wikipedia.org/wiki/Scrum_(software_development)
https://en.wikipedia.org/wiki/Static_program_analysis
https://www.educba.com/software-development/software-development-tutorials/software-engineering-tutorial/
https://www.geeksforgeeks.org/software-engineering/
https://www.javatpoint.com/advantages-and-disadvantages-of-software-engineering
https://www.javatpoint.com/software-engineering
https://www.scaler.com/topics/software-engineering/

https://en.wikipedia.org/wiki/Artificial_intelligence , https://en.wikipedia.org/wiki/List_of_programming_languages_for_artificial_intelligence
https://en.wikipedia.org/wiki/Applications_of_artificial_intelligence
https://en.wikipedia.org/wiki/AI-assisted_software_development
https://en.wikipedia.org/wiki/Automatic_programming
https://en.wikipedia.org/wiki/Low-code_development_platform
https://en.wikipedia.org/wiki/No-code_development_platform
https://en.wikipedia.org/wiki/Flow-based_programming
https://en.wikipedia.org/wiki/Vibe_coding

https://www.statista.com/markets/418/topic/484/software/#overview
https://www.statista.com/topics/1694/app-developers/
https://www.statista.com/topics/9187/open-source-software/
https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/
https://www.statista.com/statistics/793840/worldwide-developer-survey-most-used-frameworks/

https://medium.com/@osamafannan5/what-is-context-in-coding-62196776f8ac

https://awesome-architecture.com/
https://mehdihadeli.github.io/awesome-go-education/ddd/
https://github.com/mehdihadeli
https://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)
https://www.microsoft.com/en-us/research/wp-content/uploads/2019/04/devtime-preprint-TSE19.pdf
https://www.vssut.ac.in/lecture_notes/lecture1428551142.pdf , https://tnou.ac.in/NAAC/SSR/C1/1.1.5/MCA-07.pdf
https://www.tutorialspoint.com/software_engineering/index.htm , https://en.wikipedia.org/wiki/Outline_of_software_engineering
https://intellipaat.com/blog/what-is-software-engineering/ , https://www.techinterviewhandbook.org/software-engineering-interview-guide/

AI agents:
https://www.agno.com/ , https://github.com/openai/swarm , https://www.crewai.com/ , https://github.com/microsoft/autogen , https://n8n.io/ , https://rasa.com/ , https://www.langchain.com/langgraph , https://www.langflow.org/ , https://strandsagents.com/ , https://smolagents.org/ , https://flowiseai.com/ , https://github.com/langgenius/dify , https://github.com/Significant-Gravitas/AutoGPT , https://github.com/botpress/botpress , https://ai.pydantic.dev/