SOFTWARE QUALITY, PROCESS IMPROVEMENT AND TECHNOLOGY TRENDS

Today, software quality remains an issue, but who is to blame? Customers blame developers, arguing that sloppy practices lead to low-quality software. Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated. Who's right? *Both* —and that's the problem.

What is Quality?

Quality can be described from five different points of view.

(1) The ***transcendental view*** argues that quality is something you immediately recognize, but cannot explicitly define.
(2) The ***user view*** sees quality in terms of an end user's specific goals. If a product meets those goals, it exhibits quality.
(3) The ***manufacturer's view*** defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality.
(4) The ***product view*** suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product.
(5) The ***value-based view*** measures quality based on how much a customer is willing to pay for a product.

In reality, quality encompasses all of these views and more.
Quality of design refers to the characteristics that designers specify for a product. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, the design quality of a product increases if the product is manufactured according to specifications. In software development, quality of design encompasses the degree to which the design meets the functions and features specified in the requirements model. ***Quality of conformance*** focuses on the degree to which the implementation follows the design and the resulting system meets its requirements and performance goals.

User satisfaction = compliant product + good quality + delivery within budget and schedule

–A product's quality is a function of how much it changes the world for the better.‖

SOFTWARE QUALITY

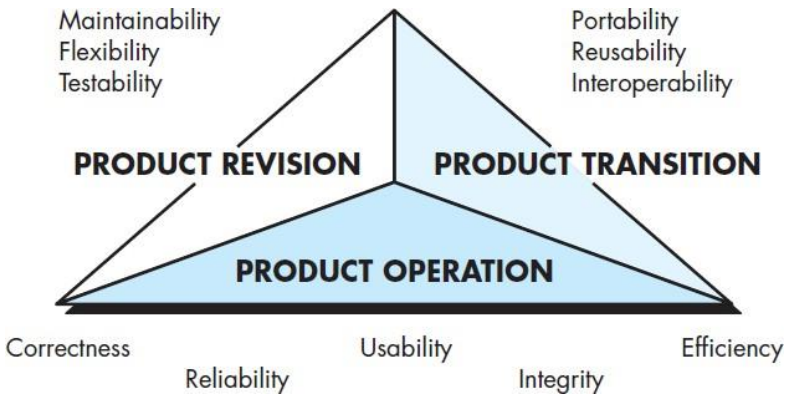
Software quality can be defined as: An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.

Garvin's Quality Dimensions

There are 8 dimensions of quality:

(1) **Performance Quality.** Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end user?
(2) **Feature quality.** Does the software provide features that surprise and delight first-time end users?
(3) **Reliability.** Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?
(4) **Conformance.** Does the software conform to local and external software standards that are relevant to the application?
(5) **Durability.** Will changes cause the error rate or reliability to degrade with time?
(6) **Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period?
(7) **Aesthetics.** Each of us has a different and very subjective vision of what is aesthetic. An aesthetic entity has certain elegance, a unique flow, and an obvious ―presence‖ that are hard to quantify but are evident nonetheless. Aesthetic software has these characteristics.
(8) **Perception.** In some situations, you have a set of prejudices that will influence your perception of quality. For example, if you are introduced to a software product that was built by a vendor who has produced poor quality in the past, your guard will be raised and your perception of the current software product quality might be influenced negatively. Similarly, if a vendor has an excellent reputation, you may perceive quality, even when it does not really exist.

McCall's Quality Factors



- *Reliability.* The extent to which a program can be expected to perform its intended function with required precision.
- *Efficiency.* The amount of computing resources and code required by a program to perform its function.
- *Integrity.* Extent to which access to software or data by unauthorized persons can be controlled.
- *Usability.* Effort required to learn, operate, prepare input for, and interpret output of a program.
- *Maintainability.* Effort required to locate and fix an error in a program.
- *Flexibility.* Effort required to modify an operational program.
- *Testability.* Effort required to test a program to ensure that it performs its intended function.
- *Portability.* Effort required to transfer the program from one hardware and/or software system environment to another.
- *Reusability.* Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.
- *Interoperability.* Effort required to couple one system to another.

ISO 9126 Quality Factors

ISO 9126 standard identifies six key quality attributes:

(1) **Functionality.** The degree to which the software satisfies stated needs as indicated by the following subattributes: suitability, accuracy, interoperability, compliance, and security.
(2) **Reliability.** The amount of time that the software is available for use as indicated by the following subattributes: maturity, fault tolerance, recoverability.
(3) **Usability** . The degree to which the software is easy to use as indicated by the following subattributes: understandability, learnability, operability.
(4) **Efficiency.** The degree to which the software makes optimal use of system resources as indicated by the following subattributes: time behavior, resource behavior.
(5) **Maintainability.** The ease with which repair may be made to the software as indicated by the following subattributes: analyzability, changeability, stability, testability.
(6) **Portability.** The ease with which the software can be transposed from one environment to another as indicated by the following subattributes: adaptability, installability, conformance, replaceability.

THE SOFTWARE QUALITY DILEMMA

If you produce a software system that has terrible quality, you lose because no one will want to buy it. If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it‘s going to take so long to complete and it will be so expensive to produce that you‘ll be out of business anyway. So

people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away.

“Good Enough” Software:
What is ‒good enough‖? Good enough software delivers high-quality functions and features that end users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs. The software vendor hopes that the vast majority of end users will overlook the bugs because they are so happy with other application functionality.

The Cost of Quality:
We know that quality is important, but it costs us time and money—too much time and money to get the level of software quality we really want. The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities and the downstream costs of lack of quality. The cost of quality can be divided into costs associated with prevention, appraisal, and failure.

- ***Prevention costs*** include (1) the cost of management activities required to plan and coordinate all quality control and quality assurance activities, (2) the cost of added technical activities to develop complete requirements and design models, (3) test planning costs, and (4) the cost of all training associated with these activities.
- ***Appraisal costs*** include activities to gain insight into product condition the ‒first time through‖ each process. Examples of appraisal costs include: (1) the cost of conducting technical reviews for software engineering work products, (2) the cost of data collection and metrics evaluation, and (3) the cost of testing and debugging
- ***Failure costs*** are those that would disappear if no errors appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. ***Internal failure costs*** are incurred when you detect an error in a product prior to shipment. Internal failure costs include: (1) the cost required to perform rework (repair) to correct an error, (2) the cost that occurs when rework inadvertently generates side effects that must be mitigated, and (3) the costs associated with the collection of quality metrics that allow an organization to assess the modes of failure. ***External failure costs*** are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labour costs associated with warranty work.

Risks:
Low-quality software increases risks for both the developer and the end user.

Negligence and Liability:
Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad. The system is late, fails to deliver desired features and functions, is

error-prone, and does not meet with customer approval. In most cases, the customer claims that the developer has been negligent and is therefore not entitled to payment. The developer often claims that the customer has repeatedly changed its requirements and has subverted the development partnership in other ways. In every case, the quality of the delivered system comes into question.

Quality and Security:

As the criticality of Web-based and mobile systems grows, application security has become increasingly important. To build a secure system, you must focus on quality, and that focus must begin during design.

The Impact of Management Actions:

As each project task is initiated, a project leader will make decisions that can have a significant impact on product quality.

- **Estimation decisions.** A software team is rarely given the luxury of providing an estimate for a project *before* delivery dates are established and an overall budget is specified. Instead, the team conducts a ―sanity check‖ to ensure that delivery dates and milestones are rational. As a consequence, shortcuts are taken, activities that lead to higher-quality software may be skipped, and product quality suffers. If a delivery date is irrational, it is important to hold your ground. Explain why you need more time, or alternatively, suggest a subset of functionality that can be delivered (with high quality) in the time allotted.
- **Scheduling decisions.** When a software project schedule is established, tasks are sequenced based on dependencies. For example, because component **A** depends on processing that occurs within components B, C, and D, component A cannot be scheduled for testing until components B, C, and D are fully tested. A project schedule would reflect this. But if time is very short, and **A** must be available for further critical testing, you might decide to test **A** without its subordinate components (which are running slightly behind schedule), so that you can make it available for other testing that must be done before delivery. After all, the deadline looms. As a consequence, **A** may have defects that are hidden, only to be discovered much later. Quality suffers.
- **Risk-oriented decisions.** Risk management is one of the key attributes of a successful software project.

ACHIEVING SOFTWARE QUALITY

Software quality doesn‘t just appear. It is the result of good project management and solid software engineering practice.

Management and practice are applied within the context of four broad activities that help a software team achieve high software quality:

1. Software engineering methods
2. Project management techniques
3. Quality control actions
4. Software quality assurance.

Software Engineering Methods

If you expect to build high-quality software, you must understand the problem to be solved. You must also be capable of creating a design that conforms to the problem while at the same time exhibiting characteristics that lead to software that exhibits the quality dimensions. There are a wide array of concepts and methods for that. If you apply those concepts and adopt appropriate analysis and design methods, the likelihood of creating high-quality software will increase substantially.

Project Management Techniques

Poor management decisions can impact the quality of the project. Software quality can be improved if:

(1) a project manager uses estimation to verify that delivery dates are achievable
(2) schedule dependencies are understood and the team resists the temptation to use shortcuts
(3) risk planning is conducted

Quality Control

Quality control encompasses a set of software engineering actions that help to ensure that each work product meets its quality goals. A combination of measurement and feedback allows a software team to tune the process when any of these work products fail to meet quality goals.

Quality Assurance

Quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions. The goal of quality assurance is to provide management and technical staff with the data necessary to be informed about product quality, thereby gaining insight and confidence that actions to achieve product quality are working.

ELEMENTS OF SOFTWARE QUALITY ASSURANCE

Software quality assurance (SQA) is often known as Quality Management. It encompasses a broad range of concerns and activities (also known as the elements of SQA).

Standards. The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.

Reviews and audits. Technical reviews are a quality control activity performed by software engineers for software engineers. Their intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

Testing. Software testing is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.

Error/defect collection and analysis. SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

Change management. If change is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices have been instituted.

Education. Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders.

Vendor management. Three categories of software are acquired from external software vendors— *shrink-wrapped packages* (e.g., Microsoft Office), a *tailored shell* that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and *contracted software* that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices that the vendor should follow (when possible), and incorporating quality mandates as part of any contract with an external vendor.

Security management. SQA ensures that appropriate process and technology are used to achieve software security.

Safety. Because software is almost always a pivotal component of human-rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.

Risk management. The SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

SQA TASKS (What is the role of SQA group?)

The Software Engineering Institute (SEI) recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that:

Prepares an SQA plan for a project. The plan is developed as part of project planning and is reviewed by all stakeholders. Quality assurance activities performed are governed by the plan. The plan identifies evaluations to be performed, audits and reviews to be conducted, standards that are applicable to the project, procedures for error reporting and tracking, work products that are produced by the SQA group, and feedback that will be provided to the software team.

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or software engineering work products.

Records any noncompliance and reports to senior management.
Noncompliance items are tracked until they are resolved.

SQA Goals:

Requirements quality. The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

Design quality. Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements. SQA looks for attributes of the design that are indicators of quality.

Code quality. Source code and related work products must conform to local coding standards and exhibit characteristics that will facilitate maintainability. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

Quality control effectiveness. A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result. SQA analyses the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

SOFTWARE PROCESS IMPROVEMENT (SPI)

The term *software process improvement* (SPI) implies many things. First, it implies that elements of an effective software process can be defined in an effective manner; second, that an existing organizational approach to software development can be assessed against those elements; and third, that a meaningful strategy for improvement can be defined. In short, SPI implies a defined software process, an organizational approach, and a strategy for improvement. The SPI strategy transforms the existing approach to software development into something that is more focused, more repeatable, and more reliable (in terms of the quality of the product produced and the timeliness of delivery).

Approaches to SPI

An organization can choose one of the many SPI frameworks. An ***SPI framework*** defines:

1. a set of characteristics that must be present if an effective software process is to be achieved
2. a method for assessing whether those characteristics are present
3. a mechanism for summarizing the results of any assessment
4. a strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing.

An SPI framework assesses the ―maturity‖ of an organization‘s software process and provides a qualitative indication of a maturity level.

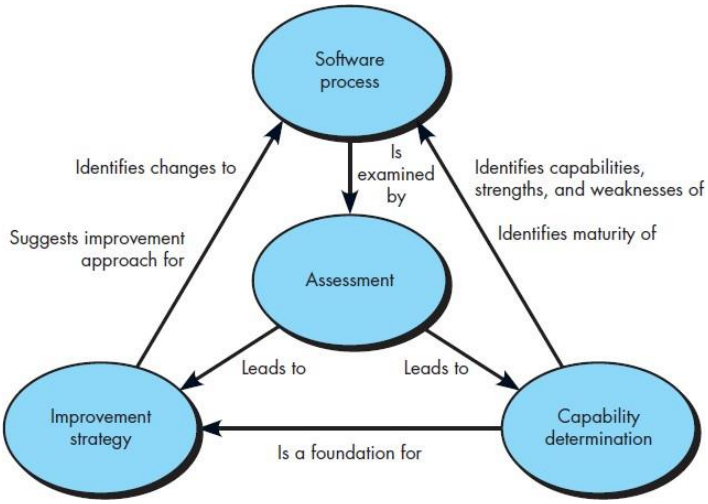


Fig: Elements of a SPI framework

There is no universal SPI framework. The SPI framework that is chosen by an organization reflects the constituency that is championing the SPI effort.

There are six different SPI support constituencies:

1. **Quality certifiers:** Process improvement efforts championed by this group focus on the following relationship:

   Quality (*Process*) ⇒ Quality (*Product*)

   Their approach is to emphasize assessment methods and to examine a well-defined set of characteristics that allows them to determine whether the process exhibits quality. They are most likely to adopt a process framework such as the CMMI, SPICE, TickIT, or Bootstrap.

2. **Formalists.** This group wants to understand process workflow. To accomplish this, they use process modelling languages (PMLs) to create a model of the existing process and then design extensions or modifications that will make the process more effective.

3. **Tool advocates.** This group insists on a tool-assisted approach to SPI

4. **Practitioners.** This constituency uses a **pragmatic approach,** –emphasizing mainstream project-, quality- and product management, applying project-level planning and metrics, but with little formal process modelling or enactment support‖

5. **Reformers.** The goal of this group is organizational change that might lead to a better software process. They tend to focus more on human issues and emphasize measures of human capability.

6. **Ideologists.** This group focuses on the suitability of a particular process model for a specific application domain or organizational structure. Rather than typical software process models (e.g., iterative models), ideologists would have a greater interest in a process that would support reuse or reengineering.

Maturity Models

A *maturity model* is applied within the context of an SPI framework. The intent of the maturity model is to provide an overall indication of the ―process maturity‖ exhibited by a software organization. That is, an indication of the quality of the software process, the degree to which practitioners understand and apply the process.

There are four levels of organizational Immaturity.

Level 0, ***Negligent*** —Failure to allow successful development process to succeed.

Level 1, ***Obstructive*** —Counterproductive processes are imposed.

Level 2, ***Contemptuous*** —Disregard for good software engineering practices, separation between software development activities and software process improvement activities, lack of a training program.

Level 3, ***Undermining*** —Total neglect of own charter, conscious discrediting of peer organizations software process improvement efforts. These activities reward failure and poor performance.

THE SPI PROCESS

The Software Engineering Institute has developed IDEAL— ―an organizational improvement model that serves as a road map for SPI activities. There are five SPI activities:

1. Assessment and Gap analysis
2. Education and Training
3. Selection and Justification
4. Installation/Migration
5. Evaluation

1. Assessment and Gap analysis

Assessment: The intent of assessment is to uncover both strengths and weaknesses in the way your organization applies the existing software process and the software engineering practices that populate the process. Assessment examines a wide range of actions and tasks that will lead to a high-quality process. As the process assessment is conducted, you should also focus on the following issues:

- **Consistency.** Are important activities, actions, and tasks applied consistently across all software projects and by all software teams?

- **Sophistication.** Are management and technical actions performed with a level of sophistication that implies a thorough understanding of best practice?
- **Acceptance.** Is the software process and software engineering practice widely accepted by management and technical staff?
- **Commitment.** Has management committed the resources required to achieve consistency, sophistication, and acceptance?

Gap analysis: The difference between local application and best practice represents a ―gap‖ that offers opportunities for improvement.

2. Education and Training

A key element of any SPI strategy is education and training for practitioners, technical managers, and more senior managers. Three types of education and training should be conducted:

- Generic software engineering concepts and methods
- Specific technology and tools
- Communication and quality-oriented topics.

3. Selection and Justification

Once the initial assessment activity has been completed and education has begun, a software organization should begin to make choices.

- First, you should choose the process model that best fits your organization, its stakeholders, and the software that you build.
- You should decide which of the set of framework activities will be applied, the major work products that will be produced, and the quality assurance checkpoints that will enable your team to assess progress.
- Next, develop an adaptable work breakdown for each framework activity, defining the task set that would be applied for a typical project.

4. Installation/Migration

Installation is the first point at which a software organization feels the effects of changes implemented as a consequence of the SPI road map. In some cases, an entirely new process is recommended for an organization. In other cases, changes associated with SPI are relatively minor, representing small, but meaningful modifications to an existing process model. Such changes are often referred to as ***process migration***.

Installation and migration are actually ***software process redesign*** **(SPR)** activities. SPR is concerned with identification, application, and refinement of new ways to dramatically improve and transform software processes. When a formal approach to SPR is initiated, three different process models are considered:

(1) the existing (―as is‖) process

(2) a transitional (–here to there‖) process
(3) the target (–to be‖) process.

5. Evaluation

The evaluation activity assesses:

- the degree to which changes have been instantiated and adopted
- the degree to which such changes result in better software quality or other tangible process benefits
- the overall status of the process and the organizational culture as SPI activities proceed.

Both qualitative factors and quantitative metrics are considered during the evaluation activity. From a qualitative point of view, past management and practitioner attitudes about the software process can be compared to attitudes polled after installation of process changes. Quantitative metrics are collected from projects that have used the transitional or –to be‖ process and compared with similar metrics that were collected for projects that were conducted under the –as is‖ process.

Risk Management for SPI:

SPI is a risky undertaking. A software organization should manage risk at three key points in the SPI process:

1. Prior to the initiation of the SPI road map
2. During the execution of SPI activities (assessment, education, selection, installation)
3. During the evaluation activity that follows the instantiation of some process characteristic.

In general, the following categories can be identified for SPI risk factors:

- budget and cost
- content and deliverables
- culture
- maintenance of SPI deliverables
- mission and goals
- organizational management
- organizational stability
- process stakeholders
- schedule for SPI development, SPI development environment, SPI development process, SPI project management, and SPI staff.

Within each category, a number of generic risk factors can be identified. For example, some of the generic risk factors defined for the culture category are:

- Attitude toward change, based on prior efforts to change
- Ability of organization members to manage meetings effectively
- Experience with quality programs, level of success

Using the risk factors and generic attributes as a guide, a risk can be developed to isolate those risks that warrant further management attention.

The CMMI

CMMI: *Capability Maturity Model Integration*
Or the SEI-CMM (Software Engineering Institute – Capability Maturity Integration)

The CMMI represents a process meta-model in two different ways:

1. as a ―continuous‖ model
2. as a ―staged‖ model.

The **continuous CMMI meta-model** describes a process in two dimensions as illustrated in below figure.

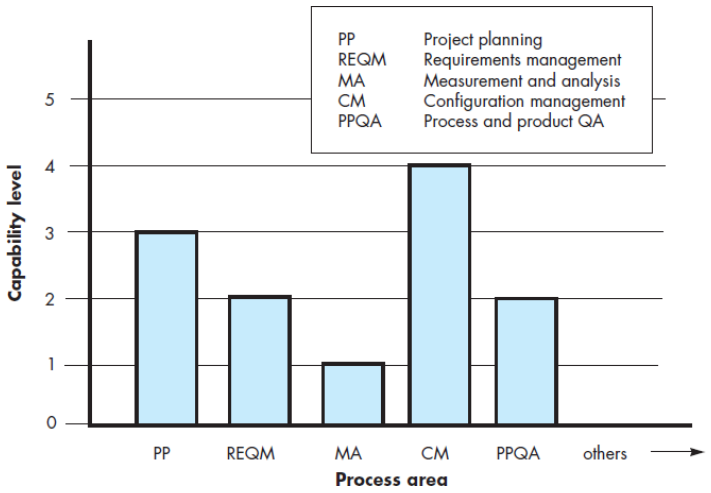


Fig: CMMI process capability profile

Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:
Level 0: *Incomplete* — The process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.
Level 1: *Performed* — All of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.
Level 2: *Managed* — All capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are ―monitored, controlled, and reviewed; and are evaluated for adherence to the process description‖.

Level 3: ***Defined*** — All capability level 2 criteria have been achieved. In addition, the process is ―tailored from the organization‘s set of standard processes‖
Level 4: ***Quantitatively managed*** — All capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment.
Level 5: ***Optimized*** — All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

The CMMI defines each process area in terms of ―specific goals‖ and the ―specific practices‖ required to achieve these goals. ***Specific goals*** establish the characteristics that must exist if the activities implied by a process area are to be effective. ***Specific practices*** refine a goal into a set of process-related activities.

For example, **project planning** is one of eight process areas defined by the CMMI for ―project management‖ category. The specific goals (SG) and the associated specific practices (SP) defined for **project planning** are:

SG 1 Establish Estimates
SP 1.1-1 Estimate the Scope of the Project
SP 1.2-1 Establish Estimates of Work Product and Task Attributes
SP 1.3-1 Define Project Life Cycle
SP 1.4-1 Determine Estimates of Effort and Cost

SG 2 Develop a Project Plan
SP 2.1-1 Establish the Budget and Schedule
SP 2.2-1 Identify Project Risks
SP 2.3-1 Plan for Data Management
SP 2.4-1 Plan for Project Resources
SP 2.5-1 Plan for Needed Knowledge and Skills
SP 2.6-1 Plan Stakeholder Involvement
SP 2.7-1 Establish the Project Plan

The **staged CMMI model** defines the same process areas, goals, and practices as the continuous model. The primary difference is that the staged model defines five maturity levels, rather than five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved.

THE ISO 9000 QUALITY STANDARDS (ISO 9001:2000 standard)

ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered. (ISO stands for ―*International Organization for Standardization”).*

To become registered to one of the quality assurance system models contained in ISO 9000, a company‘s quality system and operations are scrutinized by third party auditors for compliance to the standard and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semi-annual surveillance audits ensure continued compliance to the standard.

The requirements delineated by ISO 9001:2008

- Management responsibility
- Quality system
- Contract review
- Design control
- Document and data control
- Product identification and traceability
- Process control
- Inspection and testing
- Corrective and preventive action
- Control of quality records
- Internal quality audits
- Training, servicing, and statistical techniques.

INFO

The ISO 9001:2008 Standard

The following outline defines the basic elements of the ISO 9001:2000 standard. Comprehensive information on the standard can be obtained from the International Organization for Standardization (**www.iso.ch**) and other Internet sources (e.g., **www.praxiom.com**).

Establish the elements of a quality management system.
- Develop, implement, and improve the system.
- Define a policy that emphasizes the importance of the system.

Document the quality system.
- Describe the process.
- Produce an operational manual.
- Develop methods for controlling (updating) documents.
- Establish methods for record keeping.

Support quality control and assurance.
- Promote the importance of quality among all stakeholders.
- Focus on customer satisfaction.
- Define a quality plan that addresses objectives, responsibilities, and authority.
- Define communication mechanisms among stakeholders.

Establish review mechanisms for the quality management system.
- Identify review methods and feedback mechanisms.
- Define follow-up procedures.

Identify quality resources including personnel, training, and infrastructure elements.

Establish control mechanisms.
- For planning.
- For customer requirements.
- For technical activities (e.g., analysis, design, testing).
- For project monitoring and management.

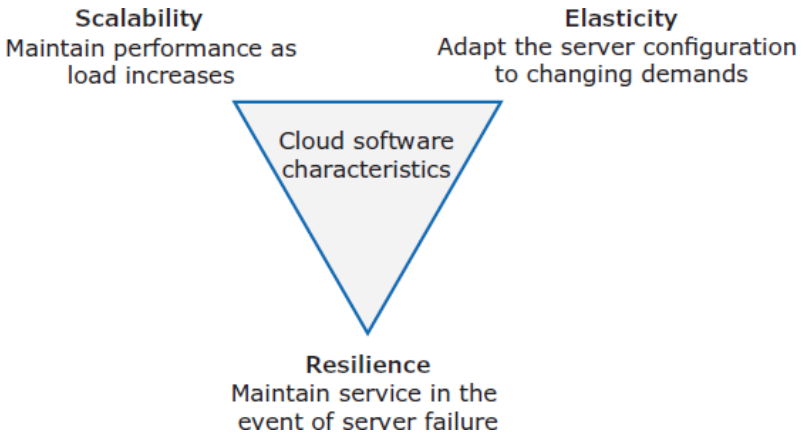
Define methods for remediation.
- Assess quality data and metrics.
- Define approach for continuous process and quality improvement.

Sr. No.	Key	ISO9000	SEI-CMM.
1	Definition	ISO9000 is an international standard of quality management and quality assurance. It certifies the companies that they are documenting the quality system elements which are needed to run a efficient and quality system.	SEI-CMM is specifically for software organizations to certify them at which level, they are following and maintaining the quality standards.
2	Focus	Focus of ISO9000 is on customer supplier relationship, and to reduce the customer's risk.	Focus of SEI-CMM is to improve the processes to deliver a quality software product to the customer.
3	Target Industry	ISO9000 is used by manufacturing industries.	SEI-CMM is used by software industry.
4	Recognition	ISO9000 is universally accepted across lots of countries.	SEI-CMM is mostly used in USA.
5	Guidelines	ISO9000 guides about concepts, principles and safeguards to be in place in a workplace.	SEI-CMM specifies what is to be followed at what level of maturity.
6	Levels	ISO9000 has one acceptance level.	SEI-CMM has five acceptance levels.
7	Validity	ISO9000 certificate is valid for three years.	SEI-CMM certificate is valid for three years as well.
8	Level	ISO9000 has no levels.	SEI-CMM has five levels, Initial, Repeatable, Defined, Managed and Optimized.

CLOUD-BASED SOFTWARE

Cloud is a very large number of remote servers that are offered for rent by companies that own these servers. You may rent a server and install your own software, or you may pay for access to software products that are available on the cloud.

The cloud servers that you rent can be started up and shut down as demand changes. This means that software that runs on the cloud can be scalable, elastic, and resilient (Figure below). These three factors (scalability, elasticity, and resilience) are the fundamental differences between cloud-based systems and those hosted on dedicated servers.



Scalability reflects the ability of your software to cope with increasing numbers of users. As the load on your software increases, the software automatically adapts to maintain the system performance and response time. Systems can be scaled by adding new servers or by migrating to a more powerful server. If a more powerful server is used, this is called **scaling up**. If new servers of the same type are added, this is called **scaling out**.

Elasticity is related to scalability but allows for **scaling down** as well as **scaling up**. That is, you can monitor the demand on your application and add or remove servers dynamically as the number of users changes.

Resilience means that you can design your software architecture to tolerate server failures. You can make several copies of your software available concurrently. If one of these fails, the others continue to provide a service.

The **benefits of adopting cloud-based approach rather than buying your own servers for software development** are shown in the below table.

Factor	Benefit
Cost	You avoid the initial capital costs of hardware procurement.
Startup time	You don't have to wait for hardware to be delivered before you can start work. Using the cloud, you can have servers up and running in a few minutes.
Server choice	If you find that the servers you are renting are not powerful enough, you can upgrade to more powerful systems. You can add servers for short-term requirements, such as load testing.
Distributed development	If you have a distributed development team, working from different locations, all team members have the same development environment and can seamlessly share all information.

Virtualization and Containers

All cloud servers are virtual servers. A virtual server runs on an underlying physical computer and is made up of an operating system plus a set of software packages that provide the server functionality required. The general idea is that a virtual server is a stand-alone system that can run on any hardware in the cloud. This ‑run anywhere‖ characteristic is possible because the virtual server has no external dependencies. An external dependency means you need some software, that you are not developing yourself. For example, if you are developing in Python, you need a Python compiler, a Python interpreter, various Python libraries, and so on.

Virtual machines (VMs) can be used to implement virtual servers (Figure below).

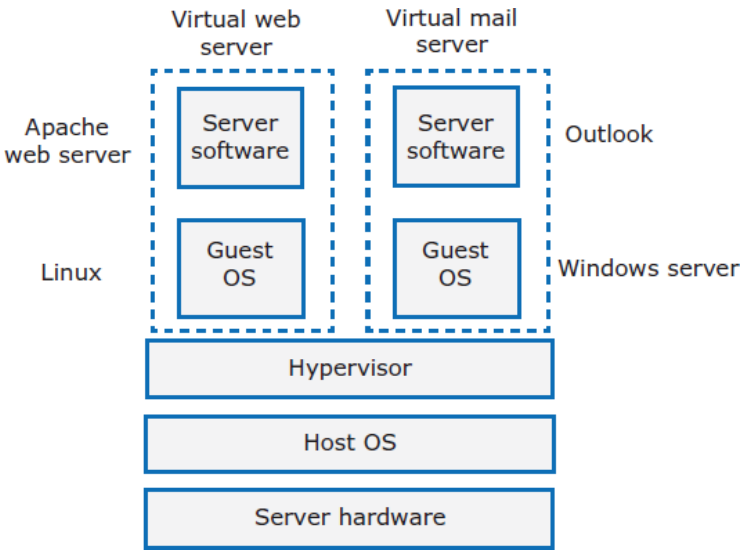


Fig: Implementing a virtual server as a virtual machine

Hypervisor provides a hardware emulation that simulates the operation of the underlying hardware. Several of these hardware emulators share the physical hardware and run in

parallel. You can run an operating system and then install server software on each hardware emulator.
The advantage of using a virtual machine to implement virtual servers is that you have exactly the same hardware platform as a physical server. You can therefore run different operating systems on virtual machines that are hosted on the same computer. For example, the above figure shows that Linux and Windows can run concurrently on separate VMs.

Why software companies use container instead of virtual machines?

If you are running a cloud-based system with many instances of applications or services, these all use the same operating system, you can use a simpler virtualization technology called _containers‘.
Containers are an operating system virtualization technology that allows independent servers to share a single operating system. They are particularly useful for providing isolated application services where each user sees their own version of an application.
Using containers dramatically speeds up the process of deploying virtual servers on the cloud. Containers are usually megabytes in size, whereas VMs are gigabytes. Containers can be started up and shut down in a few seconds rather than the few minutes required for a VM. Many companies that provide cloud-based software have now switched from VMs to containers because containers are faster to load and less demanding of machine resources. These containers are managed by the means of Dockers.

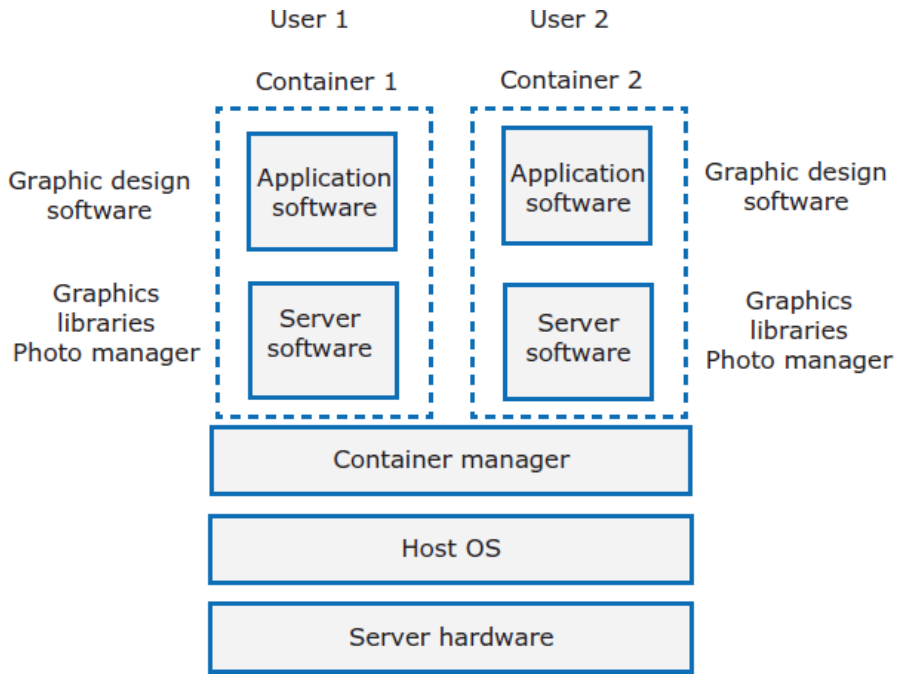


Fig: Using containers to provide isolated services

Docker:

Docker is a container management system that allows users to define the software to be included in a container as a Docker image. It also includes a run-time system that can create and manage containers using these Docker images. Below figure shows the different elements of the Docker container system and their interactions. The function of each of the elements in the Docker container system is shown the table followed.

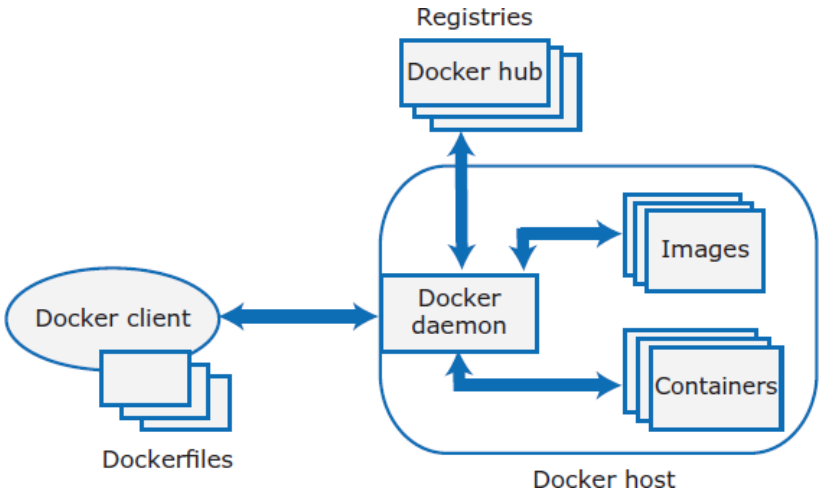


Fig: The Docker container system

Element	Function
Docker daemon	This is a process that runs on a host server and is used to set up, start, stop, and monitor containers, as well as building and managing local images.
Docker client	This software is used by developers and system managers to define and control containers.
Dockerfiles	Dockerfiles define runnable applications (images) as a series of setup commands that specify the software to be included in a container. Each container must be defined by an associated Dockerfile.
Image	A Dockerfile is interpreted to create a Docker image, which is a set of directories with the specified software and data installed in the right places. Images are set up to be runnable Docker applications.
Docker hub	This is a registry of images that has been created. These may be reused to set up containers or as a starting point for defining new images.
Containers	Containers are executing images. An image is loaded into a container and the application defined by the image starts execution. Containers may be moved from server to server without modification and replicated across many servers. You can make changes to a Docker container (e.g., by modifying files) but you then must commit these changes to create a new image and restart the container.

Table: The elements of the Docker container system

What are the benefits of containers?

1. They solve the problem of software dependencies.
    - You don‘t have to worry about the libraries and other software on the application server being different from those on your development server. Instead of shipping your product as stand-alone software, you can ship a container that includes all of the support software that your product needs.
2. They provide a mechanism for software portability across different clouds.
    - Docker containers can run on any system or cloud provider where the Docker daemon is available.
3. They provide an efficient mechanism for implementing software services and so support the development of service-oriented architectures.
4. They simplify the adoption of DevOps.

EVERYTHING AS A SERVICE

The idea of a service that is rented rather than owned is fundamental to cloud computing. Instead of owning hardware, you can rent the hardware that you need from a cloud provider. If you have a software product, you can use that rented hardware to deliver the product to your customers. In cloud computing, this has been developed into the idea of ‒everything as a service.‖

There are three levels where everything as a service is most relevant:

1. Infrastructure as a service (IaaS)
2. Platform as a service (PaaS)
3. Software as a service (SaaS)

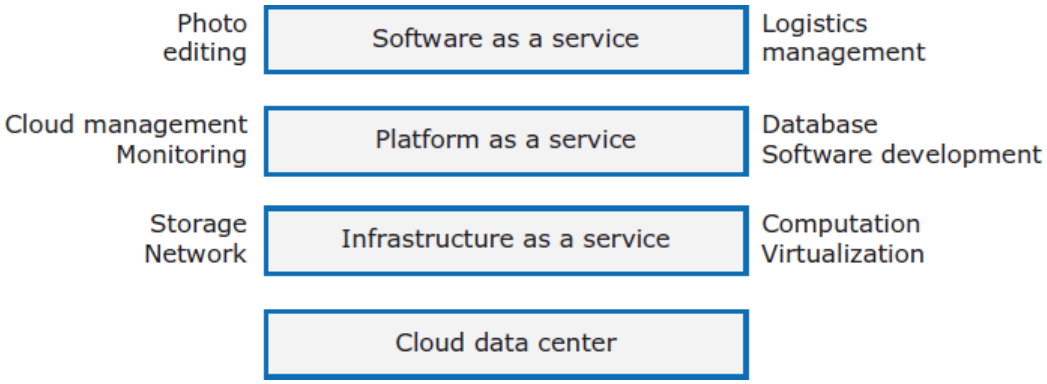


Fig: Everything as a service

Infrastructure as a service (IaaS) This is a basic service level that all major cloud providers offer. They provide different kinds of infrastructure service, such as a computer service, a network service, and a storage service. These infrastructure services may be used to implement virtual cloud-based servers. The key **benefits** of using IaaS:

- You don‘t incur the capital costs of buying hardware
- You can easily migrate your software from one server to a more powerful server
- You can add more servers if you need to as the load on your system increases.

Platform as a service (PaaS) This is an intermediate level where you use libraries and frameworks provided by the cloud provider to implement your software. These provide access to a range of functions, including SQL and NoSQL databases. Using PaaS makes it easy to develop auto-scaling software. You can implement your product so that as the load increases, additional compute and storage resources are added automatically.

Software as a service (SaaS) Your software product runs on the cloud and is accessed by users through a web browser or mobile app. This type of cloud service includes—mail services such as Gmail, storage services such as Dropbox, social media services such as Twitter, and so on.

System management responsibilities of IaaS, PaaS and SaaS

- If you are using IaaS, you have the responsibility for installing and managing the database, the system security, and the application.
- If you use PaaS, you can devolve responsibility of managing the database and security to the cloud provider.
- In SaaS, assuming that a software vendor is running the system on a cloud, the software vendor manages the application. Everything else is the cloud provider‘s responsibility.

SOFTWARE AS A SERVICE

If you deliver your software product as a service, you run the software on your servers, which you may rent from a cloud provider. Customers don‘t have to install software, and they access the remote system through a web browser or dedicated mobile app (Figure below). The payment model for SaaS is usually a subscription. Users pay a monthly fee to use the software.

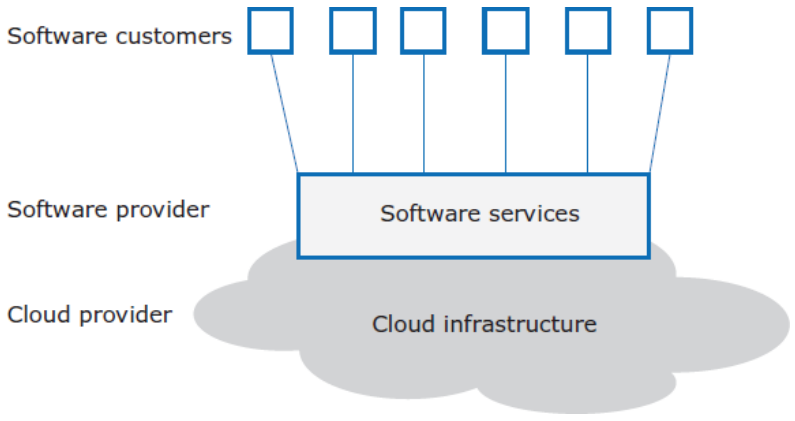


Fig: Software as a service

Many software providers deliver their software as a cloud service, but also allow users to download a version of the software so that they can work without a network connection. For example, Adobe offers the Lightroom photo management software as both a cloud service and a download that runs on the user‘s own computer. This gets around the problem of reduced performance due to slow network connections.

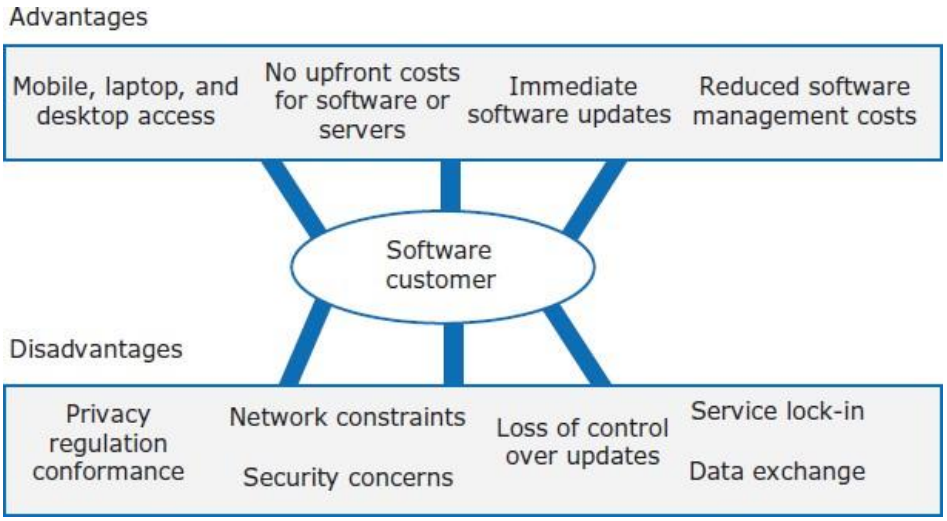
Benefits of SaaS for software product providers:

Benefit	Explanation
Cash flow	Customers either pay a regular subscription or pay as they use the software. This means you have a regular cash flow, with payments throughout the year. You don’t have a situation where you have a large cash injection when products are purchased but very little income between product releases.
Update management	You are in control of updates to your product, and all customers receive the update at the same time. You avoid the issue of several versions being simultaneously used and maintained. This reduces your costs and makes it easier to maintain a consistent software code base.
Continuous deployment	You can deploy new versions of your software as soon as changes have been made and tested. This means you can fix bugs quickly so that your software reliability can continuously improve.
Payment flexibility	You can have several different payment options so that you can attract a wider range of customers. Small companies or individuals need not be discouraged by having to pay large upfront software costs.
Try before you buy	You can make early free or low-cost versions of the software available quickly with the aim of getting customer feedback on bugs and how the product could be approved.
Data collection	You can easily collect data on how the product is used and so identify areas for improvement. You may also be able to collect customer data that allow you to market other products to these customers.

Advantages (Benefits) and **disadvantages** of SaaS for customers:

One of the most significant business benefits of using SaaS is that customers do not incur the capital costs of buying servers or the software itself. However, customers have to continue to pay, even if they rarely use the software.

The universal use of mobile devices means that customers want to access software from these devices as well as from desktop and laptop computers. People can use the software from multiple devices without having to install the software in advance. However, this may mean that software developers have to develop mobile apps for a range of platforms in order to maintain their customer base.



A further benefit of SaaS for customers is that they don‘t have to employ staff to install and update the system. However, this may lead to a loss of local expertise. A lack of expertise may make it more difficult for customers to revert to self-hosted software if they need to do so.

A characteristic of SaaS is that updates can be delivered quickly. New features are immediately available to all customers. However, customers have no control over when software upgrades are installed.

Other disadvantages of SaaS are related to storage and data management issues. These issues are given in the below table.

Issue	Explanation
Regulation	Some countries, such as EU countries, have strict laws on the storage of personal information. These may be incompatible with the laws and regulations of the country where the SaaS provider is based. If an SaaS provider cannot guarantee that their storage locations conform to the laws of the customer's country, businesses may be reluctant to use their product.
Data transfer	If software use involves a lot of data transfer, the software response time may be limited by the network speed. This is a problem for individuals and smaller companies who can't afford to pay for very high speed network connections.
Data security	Companies dealing with sensitive information may be unwilling to hand over the control of their data to an external software provider. As we have seen from a number of high-profile cases, even large cloud providers have had security breaches. You can't assume that they always provide better security than the customer's own servers.
Data exchange	If you need to exchange data between a cloud service and other services or local software applications, this can be difficult unless the cloud service provides an API that is accessible for external use.

Table: Data storage and management issues for SaaS

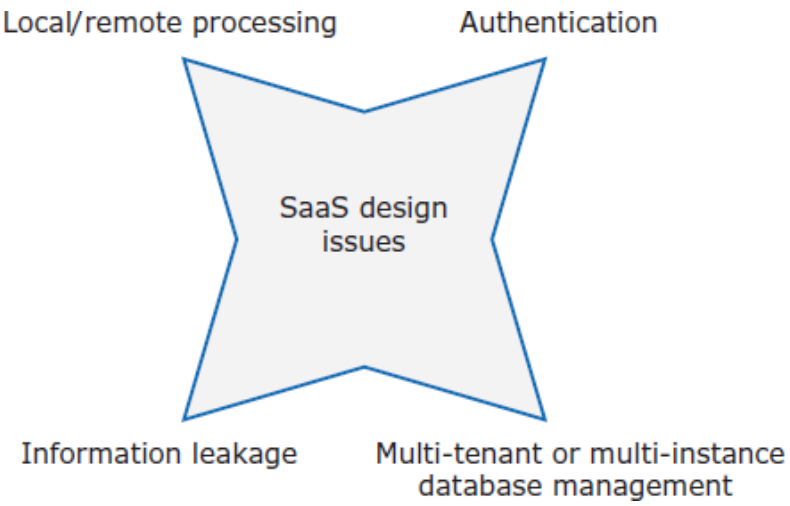
Design issues for SaaS: (The factors that you have to consider while designing the software)

A software product may be designed so that some features are executed locally in the user‘s browser or mobile app and some on a remote server. Local execution reduces network traffic and so increases user response speed. This is useful when users have a slow network connection.

On all shared systems, users have to authenticate themselves to show that they are accredited to use the system. You can set up your own authentication system, but this means users have to remember another set of authentication credentials. People don‘t like this, so for individual users, many systems allow authentication using the user‘s Google, Facebook, or LinkedIn credentials.

Information leakage is a particular risk for cloud-based software. If you have multiple users from multiple organizations, a security risk is that information leaks from one organization to another. So you need to be very careful in designing your security system to avoid it.

Multi-tenancy means that the system maintains the information from different organizations in a single repository rather than maintaining separate copies of the system and database. This can lead to more efficient operation. However, the developer has to design software so that each organization sees a virtual system that includes its own configuration and data. In a multi-instance system, each customer has their own instance of the software and its database.



MICROSERVICES ARCHITECTURE
(Microservices, Microservice Architecture, Microservice Deployment)

Software Service:
A software service is a software component that can be accessed from remote computers over the Internet. Given an input, a service produces a corresponding output, without side effects. The service is accessed through its published interface and all details of the service

implementation are hidden. The manager of a service is called the service provider, and the user of a service is called a service requestor. Services do not maintain any internal state. State information is either stored in a database or is maintained by the service requestor.

Microservices

Micro services are small-scale, stateless services that have a single responsibility. Software products that use microservices are said to have a microservices architecture. A microservices architecture is based on services that are fine-grain components with a single responsibility. If you need to create cloud-based software products, then it is recommend to use a microservice architecture.

Example of microservices

Consider a system that uses an authentication module that provides the following features:

- User registration, where users provide information about their identity, Security information, mobile (cell) phone number, and email address;
- Authentication using user ID (UID)/password;
- Two-factor authentication using code sent to mobile phone;
- User information management—for example, ability to change password or mobile phone number.

In a microservices architecture, these features are too large to be microservices. To identify the microservices that might be used in the authentication system, you need to break down the coarse-grain features into more detailed functions. Below figure shows what these functions might be for user registration and UID/password authentication.

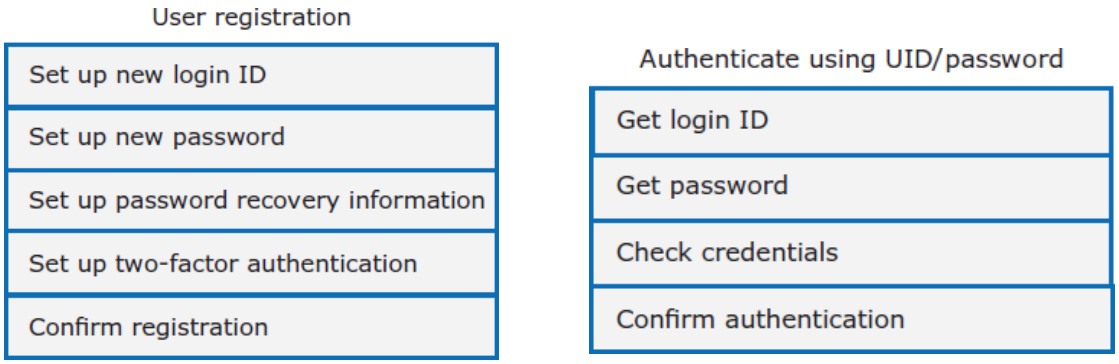


Fig: Functional breakdown of authentication features

Below figure shows the microservices that could be used to implement user authentication.

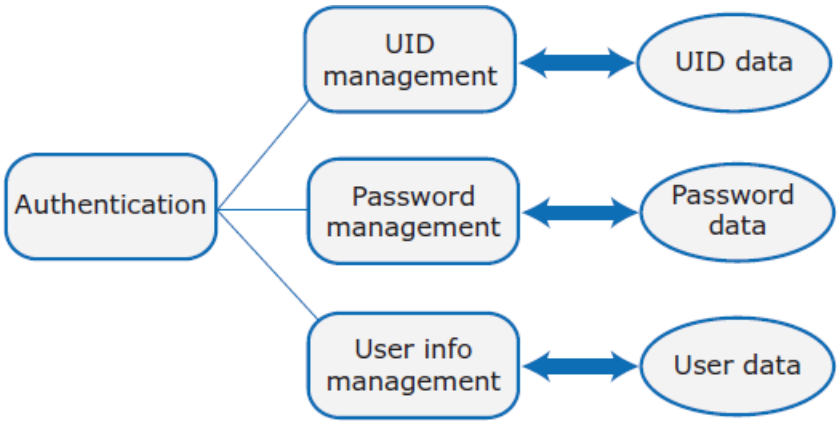


Fig: Authentication microservices

Characteristics of microservices are given below:

Characteristic	Explanation
Self-contained	Microservices do not have external dependencies. They manage their own data and implement their own user interface.
Lightweight	Microservices communicate using lightweight protocols, so that service communication overheads are low.
Implementation independent	Microservices may be implemented using different programming languages and may use different technologies (e.g., different types of database) in their implementation.
Independently deployable	Each microservice runs in its own process and is independently deployable, using automated systems.
Business-oriented	Microservices should implement business capabilities and needs, rather than simply provide a technical service.

- Microservices communicate by exchanging messages.
- A well-designed microservice should have high cohesion and low coupling.
- **Coupling** is a measure of the number of relationships that one component has with other components in the system. Low coupling means that components do not have many relationships with other components.
- **Cohesion** is a measure of the number of relationships that parts of a component have with each other. High cohesion means that all of the component parts that are needed to deliver the component‘s functionality are included in the component.
- Low coupling is important in microservices because it leads to independent services.
- High cohesion is important because it means that the service does not have to call lots of other services during execution. Calling other services involves communications overhead, which can slow down a system.

The aim of developing **highly cohesive services** has led to a fundamental principle that underlies microservice design: the **–single responsibility principle**.‖ Each element in a system should do one thing only. However, the problem with this is that –one thing only‖ is difficult to define in a way that is applicable to all services.
If you take the single responsibility principle literally, you would implement separate services for creating and changing a password and for checking that a password is correct. However, these simple services would all have to use a shared password database. This is undesirable because it increases the coupling between these services. Therefore responsibility should not always mean a single, functional activity. In this case, we can interpret a single responsibility as the responsibility to maintain stored passwords.

–Microservices‖ are small-scale components, so developers often ask **–How big should a microservice be**?‖

- It should be possible for a microservice to be developed, tested, and deployed by a service development team in two weeks or less.
- The team size should be such that the whole team can be fed by no more than two large pizzas (Amazon‘s guideline). This places an upper limit on the team size of eight to ten people (depending on how hungry they are).

The independence of microservices means that, each service has to include **support code**. These support codes are:

Microservice X

Service functionality	
Message management	Failure management
UI implementation	Data consistency management

Message management code in a microservice is responsible for processing incoming and outgoing messages. Incoming messages have to be checked for validity. Outgoing messages have to be packed into the correct format for service communication.

Failure management code in a microservice has two concerns. First, it has to cope with circumstances where the microservice cannot properly complete a requested operation. Second, if external interactions are required, such as a call to another service, it has to handle the situation where that interaction does not succeed because the external service returns an error or does not reply.

Data consistency management is needed when the data used in a microservice are also used by other services. In those cases, there needs to be a way of communicating data updates

between services and ensuring that the changes made in one service are reflected in all services that use the data.

UI implementation: For complete independence, each microservice should maintain its own user interface.

MICROSERVICES ARCHITECTURE

A microservices architecture is a tried and tested way of implementing a logical software architecture. This architectural style aims to address two fundamental problems with monolithic architecture:

1. When a monolithic architecture is used, the whole system has to be rebuilt, retested, and re-deployed when any change is made. This can be a slow process, as changes to one part of the system can adversely affect other components. Frequent application updates are therefore impossible.

2. As the demand on the system increases, the whole system has to be scaled, even if the demand is localized to a small number of system components that implement the most popular system functions.

Architectural design decisions

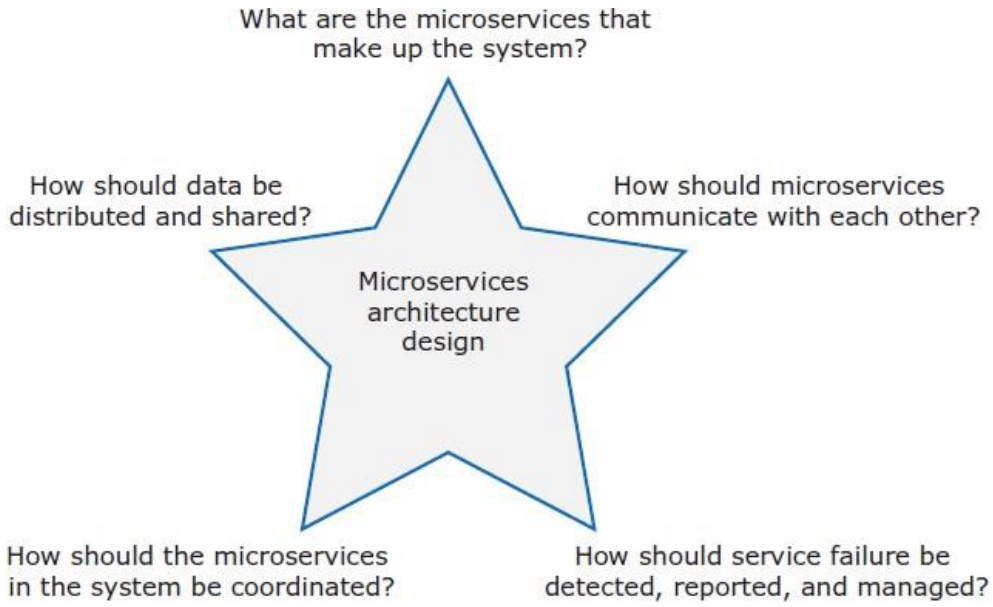


Fig: Key design issues for microservices architecture

One of the most important jobs for a system architect is to decide how the overall system should be decomposed into a set of microservices. For that there are some general guidelines:

1. ***Balance fine-grain functionality and system performance:*** If each of your services offers only a single, very specific service, however, it is inevitable that you will need to have more service communications to implement user functionality. This slows down a system.

2. ***Follow the "common closure principle":*** This means that elements of a system that are likely to be changed at the same time should be located within the same service.

3. ***Associate services with business capabilities****:* A business capability is a discrete area of business functionality that is the responsibility of an individual or a group. For example, the provider of a photo-printing system will have a group responsible for sending photos to users (dispatch capability), a set of printing machines (print capability), someone responsible for finance (payment service), and so on. You should identify the services that are required to support each business capability.

4. ***Design services so that they have access to only the data that they need***

Service communications:

Services communicate by exchanging messages. These messages include information about the originator of the message as well as the data that are the input to or output from the request.
While establishing a standard for communication, the following key decisions must be taken:

- Should service interaction be synchronous or asynchronous?
- Should services communicate directly or via message broker middleware?
- What protocol should be used for messages exchanged between services?

In **a synchronous interaction**, service A issues a request to service B. Service A then suspends processing while service B is processing the request. It waits until service B has returned the required information before continuing execution.

In **an asynchronous interaction**, service A issues the request that is queued for processing by service B. Service A then continues processing without waiting for service B to finish its computations. Sometime later, service B completes the earlier request from service A and queues the result to be retrieved by service A. Service A therefore has to check its queue periodically to see if a result is available.

Synchronous interaction is less complex than asynchronous interaction. Consequently, synchronous programs are easier to write and understand. There will probably be fewer difficult-to-find bugs. On the other hand, asynchronous interaction is often more efficient than synchronous interaction, as services are not idle while waiting for a response.

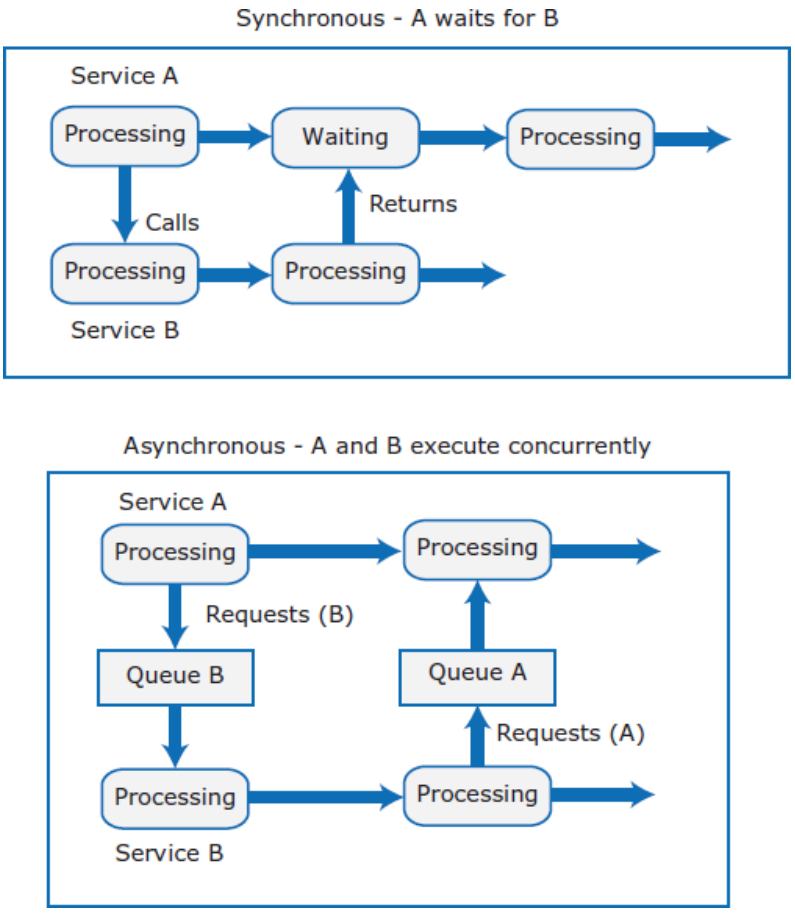


Fig: Synchronous and asynchronous microservice interaction

Direct service communication requires that interacting services know each other‘s addresses. The services interact by sending requests directly to these addresses. **Indirect communication** involves naming the service that is required and sending that request to a message broker (sometimes called a message bus). The message broker is then responsible for finding the service that can fulfill the service request. Below figure shows these communication alternatives.

Direct service communication is usually faster, but it means that the requesting service must know the URI (uniform resource identifier) of the requested service. If, that URI changes, then the service request will fail. Indirect communication requires additional software (a message broker) but services are requested by name rather than a URI. The message broker finds the address of the requested service and directs the request to it.

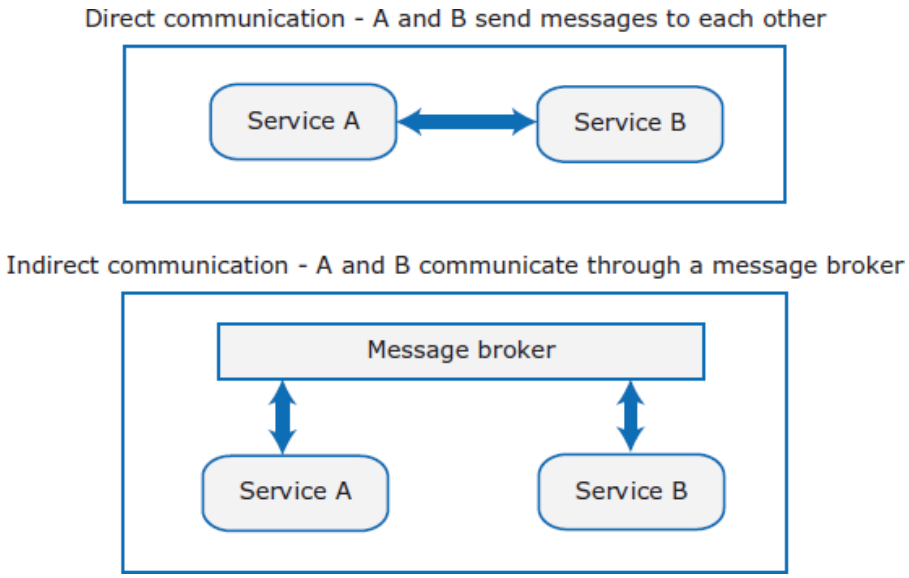


Fig: Direct and indirect service communication

Data distribution and sharing:

Each microservice should manage its own data. In an ideal world, the data managed by each service would be completely independent. There would be no need to propagate data changes made in one service to other services. However, in the real world, complete data independence is impossible.

You need to think about the microservices as an interacting system rather than as individual units. This means:

1. You should isolate data within each system service with as little data sharing as possible.
2. If data sharing is unavoidable, you should design microservices so that most sharing is read-only, with a minimal number of services responsible for data updates.
3. If services are replicated in your system, you must include a mechanism that can keep the database copies used by replica services consistent.

Access to the shared data is managed by a database management system (DBMS). Failure of services in the system and concurrent updates to shared data have the potential to cause database inconsistency.
Without controls, if services A and B are updating the same data, the value of that data depends on the timing of the updates. However, by using ACID properties, the DBMS serializes the updates and avoids inconsistency.

Systems that use microservices have to be designed to tolerate some degree of data inconsistency. The databases used by different services or service replicas need not be completely consistent all of the time.

Two types of inconsistency have to be managed:

1. ***Dependent data inconsistency*** The actions or failures of one service can cause the data managed by another service to become inconsistent.
2. ***Replica inconsistency*** Several replicas of the same service may be executing concurrently. These all have their own database copy and each updates its own copy of the service data. You need a way of making these databases ‑eventually consistent‖ so that all replicas are working on the same data.

Service coordination:

Most user sessions involve a series of interactions in which operations have to be carried out in a specific order. This is called a workflow. As an example, the workflow for UID/password authentication in which there is a limited number of allowed authentication attempts is shown in below figure.

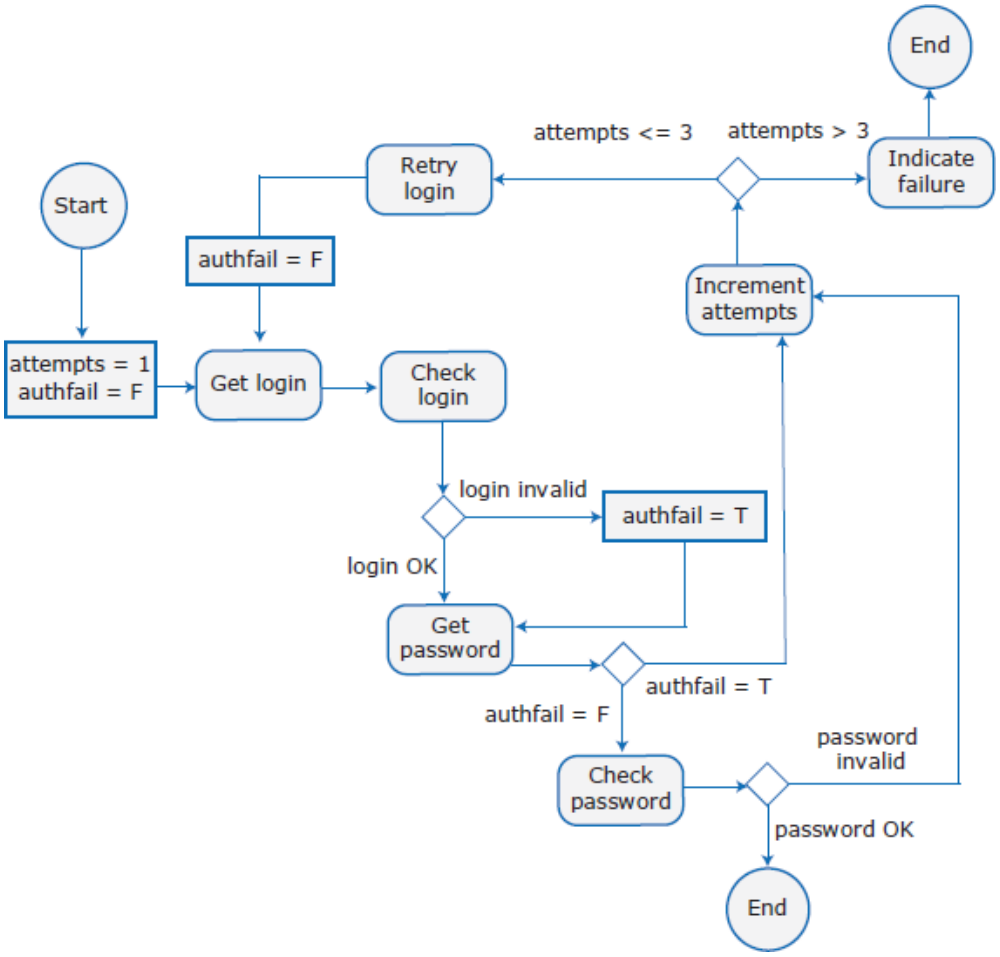


Fig: Authentication workflow

In this example workflow, the user is allowed three login attempts before the system indicates that the login has failed.

One way to **implement this workflow** is to define the workflow explicitly (either in a workflow language or in code) and to have a separate service that executes the workflow by calling the component services in turn. This is called **–orchestration**,‖ reflecting the notion that an orchestra conductor instructs the musicians when to play their parts. In an orchestrated system, there is an overall controller.

An alternative approach is called **"choreography**.‖ This term is derived from dance rather than music, where there is no –conductor‖ for the dancers. Rather, the dance proceeds as dancers observe one another. Their decision to move on to the next part of the dance depends on what the other dancers are doing.

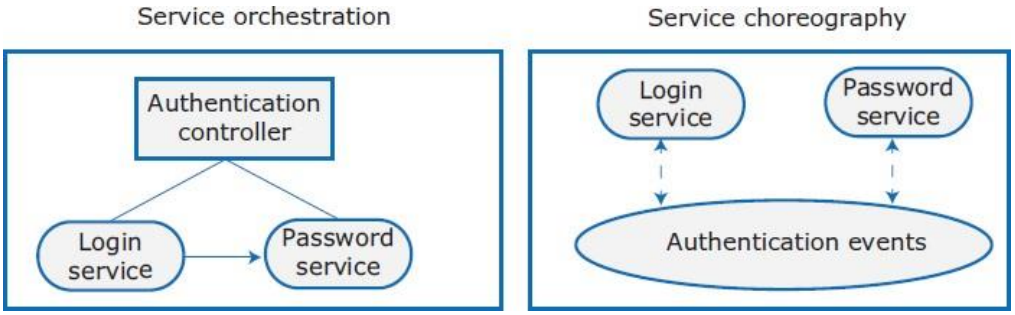


Fig: Orchestration and choreography

Choreography depends on each service emitting an event to indicate that it has completed its processing. Other services watch for events and react accordingly when events are observed. There is no explicit service controller. To implement service choreography, you need additional software such as a message broker.

A problem with service choreography is that there is no simple correspondence between the workflow and the actual processing that takes place. This makes choreographed workflows harder to debug. If a failure occurs during workflow processing, it is not immediately obvious what service has failed.

Furthermore, recovering from a service failure is sometimes difficult to implement in a choreographed system.

In an orchestrated approach, if a service fails, the controller knows which service has failed and where the failure has occurred in the overall process.

Failure management:

The three kinds of failure with in a microservices system are shown in the below table.

Failure type	Explanation
Internal service failure	These are conditions that are detected by the service and can be reported to the service requestor in an error message. An example of this type of failure is a service that takes a URL as an input and discovers that this is an invalid link.
External service failure	These failures have an external cause that affects the availability of a service. Failure may cause the service to become unresponsive and actions have to be taken to restart the service.
Service performance failure	The performance of the service degrades to an unacceptable level. This may be due to a heavy load or an internal problem with the service. External service monitoring can be used to detect performance failures and unresponsive services.

The simplest way to report microservice failures is to use HTTP status codes, which indicate whether or not a request has succeeded.

One way to discover whether a service that you are requesting is unavailable or running slowly is to put a timeout on the request. A timeout is a counter that is associated with the service requests and starts running when the request is made. Once the counter reaches some predefined value, such as 10 seconds, the calling service assumes that the service request has failed and acts accordingly.

SERVICE DEPLOYMENT (MICROSERVICE DEPLOYEMENT)

After a system has been developed and delivered, it has to be deployed on servers, monitored for problems, and updated as new versions become available. The development team is responsible for deployment and service management as well as software development. That is, in microservice architecture, we use the concepts of DevOps - a combination of Development and Operations, for software deployment.

In this area, a good practise is to adopt a policy of continuous deployment. Continuous deployment means that as soon as a change to a service has been made and validated, the modified service is re-deployed.

Continuous deployment depends on automation so that as soon as a change is committed, a series of automated activities is triggered to test the software. If the software ‒passes‖ these tests, it then enters another automation pipeline that packages and deploys the software. **Containers** are usually the best way to package a cloud service for deployment. Below figure is a simplified diagram of the continuous deployment process.

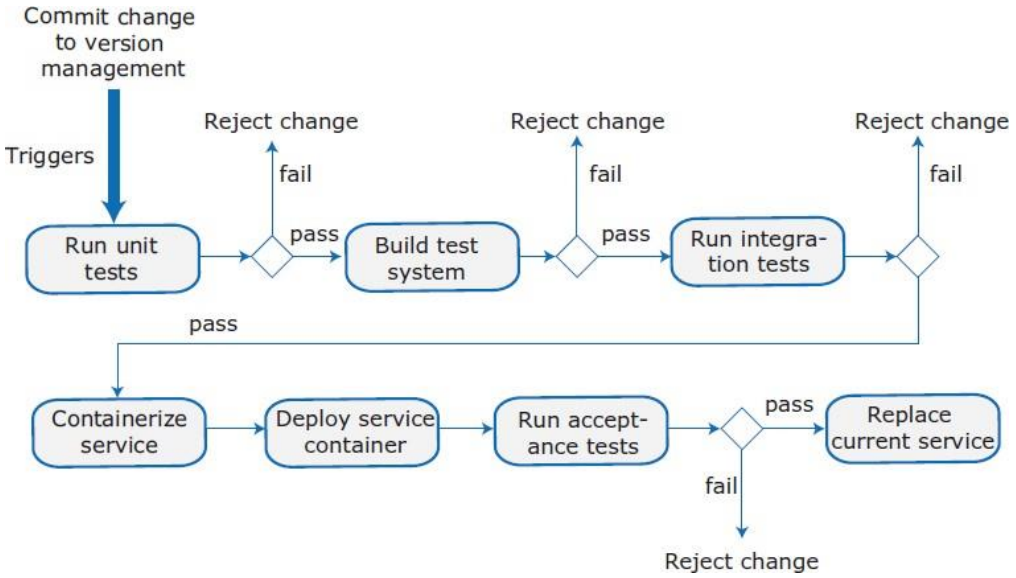


Fig: A continuous deployment pipeline