MODULE 5 SOFTWARE QUALITY, PROCESS IMPROVEMENT AND TECHNOLOGY TRENDS

SOFTWARE QUALITY

- 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.
- The definition serves to emphasize three main points:

- An *effective software process* establishes the infrastructure that supports any effort at building a high-quality software product. The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality.

- A *useful product* delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free way. A useful product always satisfies those requirements that have been explicitly stated by stakeholders. In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high quality software.

- By *adding value for both the producer and user* of a software product, high-quality software provides benefi ts for the software organization and the end-user community. The software organization gains added value because high-quality software requires less maintenance effort, fewer bug fixes, and reduced customer support. This enables software engineers to spend more time creating new applications and less on rework. The user community gains added value because the application provides a useful capability in a way that expedites some business process.

Garvin's Quality Dimensions

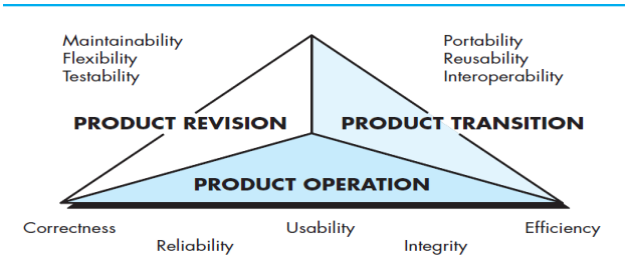
David Garvin suggests that quality should be considered by taking a multidimensional viewpoint that begins with an assessment of conformance and terminates with a transcendental (aesthetic) view. Although Garvin's eight dimensions of quality were not developed specifically for software, they can be applied when software quality is considered:

- **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?
- **Feature quality.** Does the software provide features that surprise and delight fi rst-time end users?
- **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?
- **Conformance.** Does the software conform to local and external software standards that are relevant to the application?
- **Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period? Can support staff acquire all information they need to make changes or correct defects?
- **Aesthetics.** There's no question that each of us has a different and very subjective vision of what is aesthetic. And yet, most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious "presence" that are hard to quantify but are evident nonetheless. Aesthetic software has these characteristics.
- **Perception.** In some situations, you have a set of prejudices that will influence your perception of quality.

McCall's Quality Factors

McCall, Richards, and Walters [McC77] propose a useful categorization of factors that affect software quality. These software quality factors, shown in Figure 19.1 , focus on three important aspects of a

software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments.



1. *Reliability.* The extent to which a program can be expected to perform its intended function with required precision.
2. *Efficiency.* The amount of computing resources and code required by a program to perform its function.
3. *Integrity.* Extent to which access to software or data by unauthorized persons can be controlled.
4. *Usability.* Effort required to learn, operate, prepare input for, and interpret output of a program.
5. *Maintainability.* Effort required to locate and fi x an error in a program
6. *Flexibility.* Effort required to modify an operational program.
7. *Testability.* Effort required to test a program to ensure that it performs its intended function.
8. *Portability.* Effort required to transfer the program from one hardware and/or software system environment to another.
9. *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.
10. *Interoperability.* Effort required to couple one system to another.

ISO 9216 Quality Factors

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes:

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

Targeted Quality Factors

- To conduct your assessment, you'll need to address specific, measurable (or at least, recognizable) attributes of the interface.

  - **Intuitiveness.** The degree to which the interface follows expected usage patterns so that even a novice can use it without significant training.
  - **Efficiency.** The degree to which operations and information can be located or initiated.
  - **Robustness.** The degree to which the software handles bad input data or inappropriate user interaction.
  - **Richness.** The degree to which the interface provides a rich feature set.

SOFTWARE QUALITY DILEMMA

Good Enough Software: 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: 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.

Risks: Poor quality leads to risks, some of them very serious.

Negligence and Liability:

Quality and Security:

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: software engineering methods, project management techniques, quality control actions, and 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 and factors.

- Project Management Techniques: The implications are clear: 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 so problems do not breed chaos, software quality will be affected in a positive way.

- Quality Control: Quality control encompasses a set of software engineering actions that help to ensure that each work product meets its quality goals. Models are reviewed to ensure that they are complete and consistent. Code may be inspected in order to uncover and correct errors before testing commences.

- Quality Assurance: Quality assurance establishes the infrastructure that supports solid software engineering methods, rational project management, and quality control actions—all pivotal if you intend to build high-quality software. In addition, 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 encompasses a broad range of concerns and activities that focus on the management of software quality.

Standards. The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other stakeholders. 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.

Testing. Software testing is a quality control function that has one primary goal—to fi nd 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. The only way to improve is to measure how you're doing. 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. Change is one of the most disruptive aspects of any software project. If it 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. The SQA organization takes the lead in software process improvement and is a key proponent and sponsor of educational programs.

Vendor management. 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. With the increase in cyber crime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for Web Apps, and ensure that software has not been tampered with internally. 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. Although the analysis and mitigation of risk is the concern of software engineers, the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

SQA TASKS

The charter of the SQA group is to assist the software team in achieving a high-quality end product. The Software Engineering Institute 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 by the software engineering team and the SQA group 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 AND METRICS

The SQA activities described in the preceding section are performed to achieve a set of pragmatic goals:

Requirement's 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 (e.g., other descriptive information) 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 analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

FIGURE 21.1 Software quality goals, attributes, and metrics
Source: Adapted from [Hya96].

Goal	Attribute	Metric
Requirement quality	Ambiguity	Number of ambiguous modifiers (e.g., many, large, human-friendly)
	Completeness	Number of TBA, TBD
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement
		Time (by activity) when change is requested
	Traceability	Number of requirements not traceable to design/code
	Model clarity	Number of UML models
		Number of descriptive pages per model
		Number of UML errors
Design quality	Architectural integrity	Existence of architectural model
	Component completeness	Number of components that trace to architectural model
		Complexity of procedural design
	Interface complexity	Average number of pick to get to a typical function or content
		Layout appropriateness
	Patterns	Number of patterns used
Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors (Chapter 8)
	Understandability	Percent internal comments
		Variable naming conventions
	Reusability	Percent reused components
	Documentation	Readability index
QC effectiveness	Resource allocation	Staff hour percentage per activity
	Completion rate	Actual vs. budgeted completion time
	Review effectiveness	See review metrics (Chapter 14)
	Testing effectiveness	Number of errors found and criticality
		Effort required to correct an error
		Origin of error

SPI (SOFTWARE PROCESS IMPROVEMENT)

- The term *software process improvement* (SPI) implies many things. First, it implies that elements of an effective software process can be defi ned 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 defi ned.
- The SPI strategy transforms the existing approach to software development into something that is more focused, more repeatable, and more reliable.

- Although an organization can choose a relatively informal approach to SPI, the vast majority choose one of a number of 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, and
  4. a strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing.

SPI PROCESS

1. Assessment and Gap Analysis:

The first road map activity, called *assessment,* allows you to get your bearings. 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

The difference between local application and best practice represents a “gap” that offers opportunities for improvement. The degree to which consistency, sophistication, acceptance, and commitment are achieved indicates the amount of cultural change that will be required to achieve meaningful improvement.

2. Education and Training

It follows that a key element of any SPI strategy is education and training for practitioners, technical managers, and more senior managers who have direct contact with the software organization. Three types of education and training should be conducted: generic software engineering concepts and methods, specific technology and tools, and communication and quality-oriented topics.

3. Selection and Justification

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. If the SPI assessment activity indicates that you have specific weaknesses (e.g., you have no formal SQA functions), you should focus attention on process characteristics that will address these weaknesses directly.

Ideally, everyone works together to select various process and technology elements and moves smoothly toward the installation or migration activity. In reality, selection can be a rocky road. It is often difficult to achieve consensus among different constituencies. If the criteria for selection are established by committee, people may argue endlessly about whether the criteria are appropriate and whether a choice truly meets the criteria that have been established.

4. Installation and 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. Framework activities, software engineering actions, and individual work tasks must be defi ned and installed as part of a new software engineering culture. Such changes represent a substantial organizational and technological transition and must be managed very carefully.

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.* Today, many software organizations have a “process” in place. The problem is that it doesn’t work in an effective manner. Therefore, an incremental *migration* from one process (that doesn’t work as well as desired) to another process is a more effective strategy

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, and 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.

6. Risk Management

A software organization should manage risk at three key points in the SPI process: prior to the initiation of the SPI road map, during the execution of SPI activities (assessment, education, selection, installation), and during the evaluation activity that follows the instantiation of some process characteristic. In general, the following categories [Sta97b] 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.

THE CMMI

Capability Maturity Model Integration (CMMI) [CMM07], a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity.

The CMMI represents a process meta-model in two different ways: (1) as a "continuous" model and (2) as a "staged" model. The continuous CMMI meta-model describes a process in two dimensions. 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 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 defi ned 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 according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets" .

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. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process"

Level 5: *Optimized* — All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) 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* refi ne a goal into a set of process-related activities.

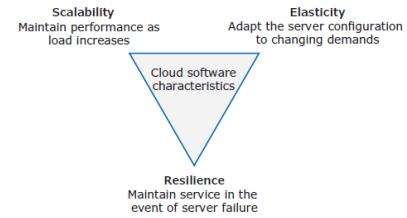
- In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence, to achieve a particular capability level, the generic goal for that level and the generic practices that correspond to that goal must be achieved

CLOUD BASED SOFTWARE

Cloud-based computing (also called Software as a Service, or SaaS) allows users access to software applications that run on shared computing resources (for example, processing power, memory, and disk storage) via the Internet. These computing resources are maintained in remote data centres dedicated to hosting various applications on multiple platforms.

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 remote servers are "virtual servers," which means they are implemented in software rather than hardware. Many virtual servers can run simultaneously on each cloud hardware node, using virtualization support that is built in to the hardware. Running multiple servers has very little effect on server performance. The hardware is so powerful that it can easily run several virtual servers at the same time.

Figure 5.1 Scalability, elasticity, and resilience



Characteristics of Cloud based Software:

1. 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. If your software is scaled out, copies of your software are created and executed on the additional servers.

2. 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. This means that you pay for only the servers you need when you need them

3. 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. You can cope with the failure of a cloud data center by locating redundant servers in different places.

Table 5.1 Benefits of using the cloud for software development

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.

- Virtual machines (VMs), running on physical server hardware, can be used to implement virtual servers (Figure 5.2). The details are complex, but you can think of the hypervisor as providing 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

- The problem with implementing virtual servers on top of VMs is that creating a VM involves loading and starting up a large and complex operating system (OS). The time needed to install the OS and set up the other software on the VM is typically between 2 and 5 minutes on public cloud providers such as AWS. This means that you cannot instantly react to changing demands by starting up and shutting down VMs.

Container Based Virtualization

- 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.
- 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.
- Containers are a lightweight mechanism for running applications in the cloud and are particularly effective for running small applications such as stand-alone services.

- From a cloud software engineering perspective, containers offer four important benefits:

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. 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. This is an approach to software support where the same team is responsible for both developing and supporting operational software

IaaS, PaaS, SaaS

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 compute 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 are that you don't incur the capital costs of buying hardware and you can easily migrate your software from one server to a more powerful server.
- You are responsible for installing the software on the server, although many preconfigured packages are available to help with this.
- Using the cloud provider's control panel, you can easily 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)

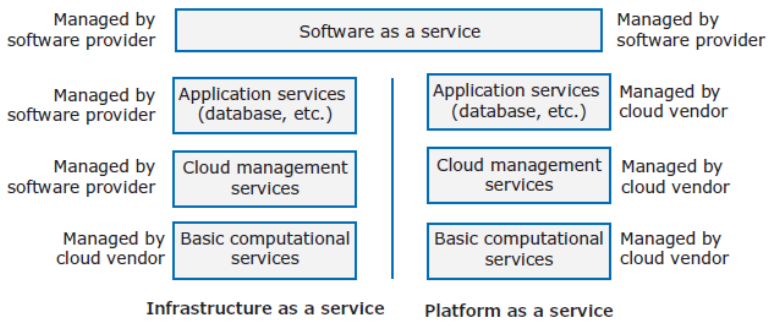
- Your software product runs on the cloud and is accessed by users through a web browser or mobile app. We all know and use this type of cloud service—mail services such as Gmail, storage services such as Dropbox, social media services such as Twitter, and so on.

- If you are running a small or medium-sized product development company, it is not cost effective to buy server hardware. If you need development and testing servers, you can implement these using infrastructure services. You can set up a cloud account with a credit card and be up and running within a few minutes. You don't have to raise capital to buy hardware, and it is easy to respond to changing demand by upscaling and downscaling your system on the cloud. You can implement your product on the cloud using PaaS and deliver your product as a software service.

- An important difference between IaaS and PaaS is the allocation of system management responsibilities. Figure 5.6 shows who has management responsibilities for SaaS, IaaS, and PaaS.

- 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.

Figure 5.6 Management responsibilities for SaaS, IaaS, and PaaS



MICRO SERVICE ARCHITECTURE

MICROSERVICES

A service should be related to a single business function. Instead of relying on a shared database and other services in the system, services should be completely independent, with their own database. They should also manage their own user interface. Replacing or replicating a service should therefore be possible without having to change any other services in the system. This type of service has become known as a "microservice." Microservices are small-scale, stateless services that have a single responsibility. Software products that use microservices are said to have a microservices architecture.

Characteristics of Microservices:

- ***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 are small-scale services that may be combined to create applications. They should be independent, so that the service interface is not affected by changes to other services. It should be possible to modify the service and re-deploy it without changing or stopping other services in the system.

- Microservices communicate by exchanging messages. A message that is sent between services includes some administrative information, a service request, and the data required to deliver the requested service.

- Cohesion and coupling are ideas that were developed in the 1970s to reflect the interdependence of components in a software system. Briefly:
  - 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. So long as you maintain its interface, you can update a service without having to change other services in the system.

- 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.

- Although microservices should focus on a single responsibility, this does not mean they are like functions that do only one thing. Responsibility is a broader concept than functionality, and the service development team has to implement all the individual functions that implement the service's responsibility.

- In addition to this functionality, the independence of microservices means that each service has to include support code that may be shared in a monolithic system.

- Message management code in a microservice is responsible for processing incoming and outgoing messages. Incoming messages have to be checked for validity, and the data extracted from the message format are used. 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

MICROSERVICES ARCHITECTURE

- A microservices architecture is not like a layered application architecture that defines the common set of components used in all applications of a particular kind.
- Rather, a microservices architecture is an *architectural style*— a tried and tested way of implementing a logical software architecture.
- For web-based applications, this architectural style is used to implement logical client-server architectures, where the server is implemented as a set of interacting microservices.
- The microservices architectural style aims to address two fundamental problems with the multi-tier software architecture for distributed systems.

- Microservices are self-contained and run in separate processes. In cloud based systems, each microservice may be deployed in its own container.

- This means a microservice can be stopped and restarted without affecting other parts of the system. If the demand on a service increases, service replicas can be quickly created and deployed. These do not require a more powerful server, so scaling out is typically much cheaper than scaling up.

Architecture Design Decisions

- In a microservice-based system, the development teams for each service are autonomous. They make their own decisions about how best to provide the service. This means the system architect should not make technology decisions for individual services; these are left to the service implementation team.

Key Design Questions

1. What are the microservices that makes the system?
2. How should microservices communicate with each other?
3. How should service failure be detected, reported and managed?
4. How should the microservices in system be coordinated?

5. How should data be distributed and shared?

General Guidelines to decompose microservices

1. Balance fine-grain functionality and system performance
2. Follow the “common closure principle”
3. Associate services with business capabilities
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. The messages that are exchanged are structured to follow a message protocol. This is a definition of what must be included in each message and how each component of the message can be identified. When you are designing a microservices architecture, you have to establish a standard for communications that all microservices should follow. Key decisions that you have to make are:

- 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?

Data Distributions and Sharing

A general rule of microservice development is that each microservice should manage its own data. 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.

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.

- An alternative approach that is often recommended for microservices 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.

- In a microservices architecture, 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 that supports a publish
- and subscribe mechanism. Publish and subscribe means that services “publish” events to other services and “subscribe” to those events that they can process.

Failure Management

Three types of failures occur in microservice systems:

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. Service responses should include a status that reflects the success or otherwise of the service request. Status code 200 means the request has been successful, and codes from 300 to 500 indicate some kind of service failure. Requests that have been successfully processed by a service should always return the 200 status code.

MICROSERVICE DEPLOYMENT

- A general principle of microservice-based development is that the service development team has full responsibility for their service, including the responsibility of deciding when to deploy new versions of that service.
- Good practice in this area is now 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 is a process in which new versions of a service are put into production as soon as a service change has been made. It is a completely automated process that relies on automated testing to check that the new version is of production quality.

- If continuous deployment is used, you may need to maintain multiple versions of deployed services so that you can switch to an older version if problems are discovered in a newly deployed service

- The deployment of a new service version starts with the programmer committing the code changes to a code management system such as Git

- Deployment involves adding the new service to a container and installing the container on a server. Automated “whole system” tests are then executed. If these system tests run successfully, the new version of the service is put into production

- Containers are usually the best way to package a cloud service for deployment. Containers are a deployment unit that can execute on different servers so that the service development team does not have to take server configuration issues into account