Software Evolution - Software Engineering

•

Software Evolution is a term that refers to the process of developing software initially, and then timely updating it for various reasons, i.e., to add new features or to remove obsolete functionalities, etc. This article focuses on discussing Software Evolution in detail.

What is Software Evolution?

The software evolution process includes fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers.

1. The cost and impact of these changes are accessed to see how much the system is affected by the change and how much it might cost to implement the change.

Example:
A **banking app** team receives a request to add **fingerprint authentication**. The team analyzes:

- **Cost:** hiring a security consultant, additional testing, and time for integration.
- **Impact:** changes in the login module, database access permissions, and app store re-approval.
  After the analysis, they estimate it will cost **$50,000** and take **4 weeks** to implement.

2. If the proposed changes are accepted, a new release of the software system is planned.

Example:
After management approves the fingerprint authentication feature, the project manager schedules it for version 3.2 of the banking app.

A release timeline is created—development starts in April, testing in May, and public release in June.

3. During release planning, all the proposed changes (fault repair, adaptation, and new functionality) are considered.

Example:
For the **version 3.2** release, the team lists all changes:

- **Fault repair:** Fix a bug causing transaction history to display incorrectly.
- **Adaptation:** Update the app to work with the new **Android 15** OS.
- **New functionality:** Add fingerprint authentication. All of these are prioritized and scheduled for the same release.

4. A design is then made on which changes to implement in the next version of the system.

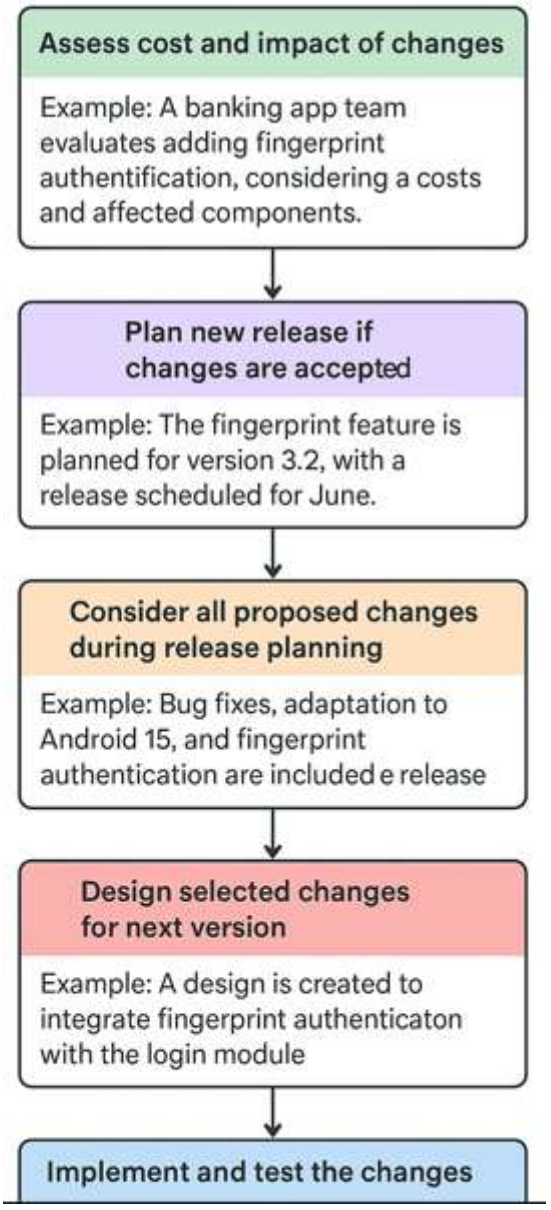
Example:
The system architect creates a **technical design document** outlining how fingerprint authentication will integrate with the existing login module using **Android's biometric API**.

The design also specifies UI updates, data encryption methods, and unit testing requirements.

5. The process of change implementation is an iteration of the development process where the revisions to the system are designed, implemented, and tested.

   Example:
   Developers code the fingerprint feature, testers verify it on multiple devices, and QA ensures that login and security remain stable.

After testing and user feedback, version **3.2** is officially released to the Play Store and App Store.


Assess cost and impact of changes
Example: A banking app team evaluates adding fingerprint authentification, considering a costs and affected components.
Plan new release if changes are accepted
Example: The fingerprint feature is planned for version 3.2, with a release scheduled for June.
Consider all proposed changes during release planning
Example: Bug fixes, adaptation to Android 15, and fingerprint authentication are included e release
Design selected changes for next version
Example: A design is created to integrate fingerprint authenticaton with the login module
Implement and test the changes

Necessity of Software Evolution

Software evaluation is necessary just because of the following reasons:

1. **Change in requirement with time:** With time, the organization's needs and modus Operandi of working could substantially be changed so in this frequently changing time the tools (software) that they are using need to change to maximize the performance.

Explanation:

With time, an organization’s business goals, processes, and customer needs evolve. The software that once met requirements may no longer support modern workflows, automation, or integration demands. Hence, continuous evolution is needed to keep up with the organization’s direction.

Real-Time Example:

Netflix originally started as a DVD rental platform, but as customer behavior shifted toward online streaming, Netflix had to evolve its entire software system — from DVD tracking systems to a cloud-based, high-performance streaming platform using AWS microservices.

This required major re-engineering of architecture, content delivery systems, and recommendation algorithms.

Visualization Suggestion:

A timeline showing Netflix’s evolution:
DVD Rentals → Online Streaming → Cloud Migration → AI-driven Recommendations.

2. **Environment change:** As the working environment changes the things(tools) that enable us to work in that environment also changes proportionally same happens in the software world as the working environment changes then, the organizations require reintroduction of old software with updated features and functionality to adapt the new environment.

Explanation:
When the technological or operational environment changes (e.g., operating systems, hardware, cloud infrastructure), the software must adapt to remain functional and compatible.

Real-Time Example:

Microsoft Office evolved from a **desktop-based suite (Office 2003)** to **cloud-integrated Office 365 (now Microsoft 365)**.

As the world moved toward **remote work and cloud collaboration**, the environment changed. Microsoft had to rebuild its software to support real-time co-authoring, automatic cloud saving, and cross-device synchronization.

Visualization Suggestion:
Old Environment: Local servers, desktop apps → **New Environment:** Cloud, web-based collaboration.

---

3. **Errors and bugs:** As the age of the deployed software within an organization increases their preciseness or impeccability decrease and the efficiency to bear the increasing complexity workload also continually degrades.

   So, in that case, it becomes necessary to avoid use of obsolete and aged software. All such obsolete Pieces of software need to undergo the evolution process in order to become robust as per the workload complexity of the current environment.

Explanation:
As software ages, its performance and accuracy may degrade. Legacy code becomes harder to maintain and may not handle modern workloads efficiently. Continuous updates are necessary to fix bugs and optimize performance.

Real-Time Example:

Windows 10 → Windows 11 upgrade was largely motivated by long-standing performance, compatibility, and memory management issues found in Windows 10.

Microsoft redesigned kernel handling and introduced **enhanced hardware security integration** (TPM 2.0 requirement) to reduce errors and improve reliability.

Visualization Suggestion:
Chart showing "bug density" and "system crashes" decreasing after evolution.

4. **Security risks:** Using outdated software within an organization may lead you to at the verge of various software-based cyberattacks and could expose your confidential data illegally associated with the software that is in use.

   So, it becomes necessary to avoid such security breaches through regular assessment of the security patches/modules are used within the software. If the software isn't robust enough to bear the current occurring Cyber attacks so it must be changed (updated).

Explanation:
Outdated software is highly vulnerable to cyberattacks. Hackers exploit unpatched vulnerabilities in old systems. Software evolution ensures regular security updates and robust defense mechanisms.

Real-Time Example:

The **2017 WannaCry ransomware attack** exploited a vulnerability in outdated **Windows XP** systems that hadn't received updates.

Organizations that had upgraded to **Windows 10** were immune. This highlighted how **failure to evolve software** exposes critical systems to major security breaches.

Visualization Suggestion:
Bar chart comparing "Security Incidents" in **Legacy Systems vs Updated Systems**.

5. **For having new functionality and features:** In order to increase the performance and fast data processing and other functionalities, an organization need to continuously evolute the software throughout its life cycle so that stakeholders & clients of the product could work efficiently.

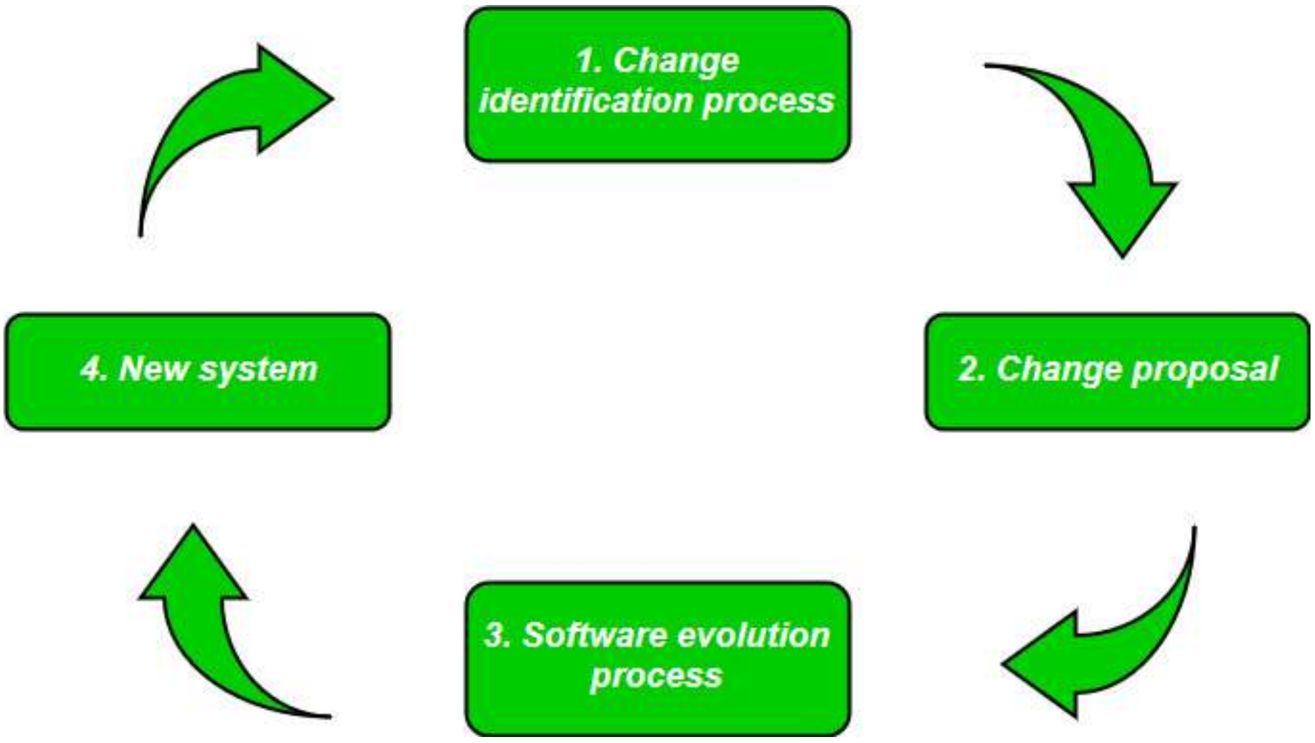
Explanation:
To stay competitive, software must continuously evolve by introducing new features, automation capabilities, or integration options to enhance performance and user satisfaction.

Real-Time Example:

Instagram initially launched as a simple **photo-sharing app**. Over time, user expectations changed, leading Instagram to evolve with features like **Stories, Reels, Shopping, and AI filters**.

These new functionalities kept Instagram relevant and engaging in an era dominated by TikTok and YouTube Shorts.

Visualization Suggestion:
Feature Evolution Timeline for Instagram:
Photo Upload → Filters → Stories → Reels → Shopping → AI Effects.


1. Change identification process
2. Change proposal
3. Software evolution process
4. New system

Software Evolution

Laws used for Software Evolution
Lehman's Laws of Software Evolution

1. Law of Continuing Change

Definition:

Any software system that represents some real-world process or reality must continually adapt and change, or it becomes progressively less useful over time.

Explanation:

As the environment, business rules, or user needs change, software must also evolve. If it doesn't, it becomes obsolete because it no longer accurately models the real-world system it supports.

Real-Time Example:

Example: Google Maps

Google Maps continuously updates routes, traffic data, new roads, and public transit changes.
If Google stopped updating its data or features, it would quickly become inaccurate and useless for navigation — especially in fast-developing cities where infrastructure changes rapidly.

Visualization Suggestion:

- Line chart showing **“Software Usefulness” decreasing** over time without updates.
- “Adaptation Curve” showing how regular updates keep the software relevant.

2. Law of Increasing Complexity

As an evolving program changes, its structure becomes more complex unless effective efforts are made to avoid this phenomenon.

Definition:

As software evolves, its internal structure becomes more complex unless deliberate efforts (like refactoring) are made to control it.

Explanation:

When developers add features or make modifications over time, dependencies grow, code coupling increases, and the architecture can degrade.
Without continuous restructuring, complexity grows — making the software harder to maintain.

Real-Time Example:

Example: Facebook (Meta)

Facebook started as a simple social network for students. As it evolved to include pages, ads, marketplace, reels, and AI moderation — its backend systems became highly complex.
Meta had to **refactor and modularize** its architecture into **microservices** to manage this complexity effectively.

Visualization Suggestion:

- Diagram showing **simple monolithic architecture → highly connected spaghetti code → modular microservices.**
- Graph showing “Complexity” rising with each version unless “Refactoring Effort” is applied.

3. Law of Conservation of Organization Stability

Over the lifetime of a program, the rate of development of that program is approximately constant and independent of the resource devoted to system development.

Definition:

Over the lifetime of a software system, the rate of development (i.e., the number of meaningful changes per release) remains roughly constant, regardless of the number of people or resources added.

Explanation:

Even if more developers or funding are added, productivity doesn’t scale proportionally because coordination, integration, and testing efforts grow with team size.

Real-Time Example:

Example: Windows Operating System

Microsoft has thousands of engineers, yet the **number of major features introduced per release** remains relatively stable. Adding more engineers doesn’t dramatically increase feature output — because integration, testing, and dependency management become bottlenecks.

Visualization Suggestion:

- Bar graph showing **resources increasing** while **rate of change (features/releases)** remains constant.
- “Effort vs Output” curve leveling off (law of diminishing returns).

4. Law of Conservation of Familiarity

This law states that during the active lifetime of the program, changes made in the successive release are almost constant.

Definition:

During the active lifetime of a system, the amount of change in each successive release is roughly constant to maintain user and developer familiarity.

Explanation:

If software changes too drastically between versions, users struggle to adapt, and developers face more bugs.
Hence, organizations introduce **incremental changes** to preserve user familiarity and stability.

Real-Time Example:

Example: Apple iOS Updates

Apple’s iOS evolves gradually. Each update introduces **incremental UI/UX changes** (like minor icon redesigns or new features) instead of complete overhauls.
This maintains **user familiarity** and avoids confusion while still improving functionality and aesthetics.

Visualization Suggestion:

- Timeline showing small, incremental updates (iOS 10 → iOS 18).
- Graph of “Change Magnitude” per release staying nearly constant.