Jumping into software development without a pre-defined plan is a recipe for overbudgeting, delays, and costly failures. Instead of rushing into a project, more and more companies are turning to SDLC strategies that enable them to ship high-quality software as quickly, safely, and cost-effectively as possible.

This article goes through **everything a company needs to know to adopt SDLC-based software development**. We explain how SDLC strategies work, dive deep into each typical phase of a product's life cycle, and present the market's most reliable SDLC methodologies.



**SDLC Meaning (Software Development Life Cycle)**

An SDLC (software development life cycle) is a big-picture breakdown of all the steps involved in software creation (planning, coding, testing, deploying, etc.). Companies define custom SDLCs to create a predictable, [iterative framework](https://phoenixnap.com/glossary/iterative-development) that guides the team through all major stages of development.

The character and the exact number of phases in an SDLC vary between businesses and projects. The most common models are variations on the following steps:

* Requirement analysis.
* In-depth planning.
* Product design.
* Coding.
* Testing.
* Deployment.
* Post-production maintenance.

An SDLC strategy enables a business to set a tried-and-tested foundation for every software-related project. Teams develop high-quality products with more speed and consistency while the company maximizes its ROI by boosting its ability to:

* Meet deadlines and keep projects within the assigned [IT budget](https://phoenixnap.com/blog/it-cost-reduction-strategy).
* Maintain high code quality standards.
* Keep bugs and [vulnerabilities](https://phoenixnap.com/blog/vulnerability-assessment) out of production.
* Align product features with business goals.
* Prioritize tasks correctly.
* Avoid scenarios in which team members work on the same, conflicting, or low-value tasks.
* Lower the number of after-the-fact fixes that impact the UX.

SDLC models are the basis of every [DevOps pipeline](https://phoenixnap.com/blog/devops-pipeline" \t "_blank). If you are considering a [transition to DevOps](https://phoenixnap.com/blog/how-to-transition-to-devops), ensure the team has a firm grasp of SDLC strategies before you introduce radical workflow changes.

**How SDLC Works**

An SDLC outlines each stage of software development, breaking down the process into separate phases that have individual:

* Goals.
* Tasks.
* Expectations.
* Process instructions.
* Documentation.
* Deliverables.
* Go-to personnel (specified either by name or position).

The exact number and nature of steps depend on the business and its product goals. On average, **most companies define SDLCs with five to seven phases**, although more complex projects reach ten or more stages.

Each step in an SDLC results in an output (document, diagram, working software, etc.) that acts as the necessary input for the next step. Despite this funnel-like approach,**modern SDLC strategies are not strictly linear**. The team often goes back a step or two in the SDLC to perform fixes or make improvements.

A product's SDLC must be a living process that the team regularly updates (or at least reviews). Keeping an SDLC up to date takes a combined effort of business analysts, developers, QA staff members, and stakeholders.

SDLC strategies have been around since the 1960s, and most of its core concepts have evolved over time. The most significant change occurred in the testing phase. Whereas testing is traditionally a separate SDLC stage, teams nowadays prefer integrating security activities throughout the life cycle to create more reliable software that's secure by design.

Ensuring every phase of the SDLC accounts for security is vital, but do not overlook the value of a dedicated testing phase. There's no reason not to have a separate stage for in-depth testing even if other SDLC steps have some built-in security analysis.

**SDLC Phases**

While every SDLC is unique, all life cycles go through similar steps. Let's take a close look at every typical phase of an average software development life cycle.



**Requirements Analysis**

The first step of any SDLC is to define the project's requirements. Some critical questions during this stage are:

* What is the goal of the new project?
* What is the business hoping to get from the product?
* Is the team going to write code from scratch, or are we upgrading an existing system?
* Do we have any hard deadlines?
* Do we have the necessary knowledge in-house, or will we have to outsource some part(s) of the project?

This stage requires a combined effort of business analytics, operations, leadership, development, and security teams. In some use cases, asking end users for input is also a valuable source of info.

All the data gathered in this stage goes into **a Software Requirement Specification (SRS) document**. An SRS file includes all the software, [hardware](https://phoenixnap.com/glossary/what-is-hardware), security, and [network](https://phoenixnap.com/blog/what-is-network-security) specifications for the upcoming product, but the file also contains info concerning:

* Resource allocation.
* Capacity planning.
* Project scheduling.
* Cost estimation.
* Provisioning.

**The output of this step:** An SRS document that defines project goals and scope, plus provides product requirements and rough project estimations (budget, resources, deadlines, etc.).

**Feasibility Study**

Senior business analysts perform a feasibility study to determine the software's viability. The usual approach is to focus primarily on these five factors:

* Budget constraints.
* Legal implications.
* Operational requirements.
* Available in-house skills.
* The required project timeframe.

Analysts add the findings of this phase to the existing SRS document, after which a team of decision-makers reviews the report to approve:

* Project plans and direction.
* Estimated costs.
* Projected schedules.
* The necessary resources.

The higher management either signs off on the project or asks the team to go back a step in the SDLC and come up with a new suggestion.

**The output of this step:** An expanded SRS document approved by higher management.

**Design Plan**

Once there is an approved project direction, the team starts creating a design plan that explains all major aspects of the new product, including its:

* [Architecture](https://phoenixnap.com/glossary/application-architecture) (programming language, [databases](https://phoenixnap.com/kb/what-is-a-database), interfaces, [operating system](https://phoenixnap.com/glossary/operating-system), pre-made templates, [APIs](https://phoenixnap.com/glossary/api-definition), etc.).
* Features list.
* [Infrastructure](https://phoenixnap.com/blog/devops-infrastructure) requirements.
* UI design.
* Necessary security measures (e.g., [SSL encryption](https://phoenixnap.com/kb/what-is-an-ssl-certificate), [password](https://phoenixnap.com/blog/strong-great-password-ideas) protection, recommended [database migrations](https://phoenixnap.com/blog/database-migration), etc.).

The team gathers this info in **the Design Document Specification (DDS)**. A stakeholder reviews the DDS and approves a direction based on the following factors:

* Design modularity.
* Risk assessment.
* Product robustness.
* Time constraints.

Some companies decide to create a prototype during this SDLC stage. While time-consuming, prototyping is much less expensive than making radical changes after the development phase.

**The output of this step:** A detailed DDS that lists all the info developers require to code the product.

**Software Development**

The development team gets familiar with the DDS and starts working on the code. Typically, this step is the most time-consuming phase of SDLC, so we recommend using [agile methodologies](https://phoenixnap.com/glossary/agile-methodology) to speed up coding.

This phase results in operational software that meets all the requirements listed in the SRS and DDS. While the code still awaits advanced testing, the team should already put the product through basic tests (such as static code analysis and code reviews for multiple device types).

**The output of this step:** Source code of a testable, fully functional software.

**In-Depth Software Testing**

The software that comes out of the previous SDLC phase now goes through extensive testing. Companies have a wide variety of [testing methods](https://phoenixnap.com/blog/software-testing-methodologies) to evaluate the new product, such as:

* Code quality testing.
* [Unit testing](https://phoenixnap.com/glossary/unit-testing) (functional tests).
* Integration testing.
* Performance testing.
* Security testing.
* Acceptance testing.
* Nonfunctional testing.

Most teams rely on [automated tests](https://phoenixnap.com/blog/automated-security-testing-best-practices) to speed up this phase, but some manual examinations are also valuable ([penetration tests](https://phoenixnap.com/blog/penetration-testing) are a good example).

If the team discovers a defect, the code goes back a step in its life cycle, and developers create a new, flaw-free version of the software. The testing stage ends when the product is stable, free of bugs, and up to quality standards defined in the previous phases.

**The output of this step:** A thoroughly tested version of the product ready for a production environment.

**Software Deployment**

The product leaves the testing phase and is ready to go into production. Some projects require the team to write user manuals or create instructional videos before the software becomes available to end users.

Ideally, the deployment phase happens automatically (typically as a part of [CI/CD](https://phoenixnap.com/blog/what-is-ci-cd)). Companies with lower maturity or in some highly regulated industries may require manual approvals during this SDLC stage.

Most companies deploy new software to a small percentage of users (10 to 15%) and slowly phase it into the rest of the customer base. Gradual introduction means you limit the impact on the UX if there's an overlooked issue with the product.

**The output of this step:** A fully functional and tested product that's available to end users.

**Product Maintenance and Enhancement**

Every shipped piece of software requires periodic reviews and updates based on user feedback. The most common activities during this stage are:

* Bug fixing.
* Setting up [continuous monitoring](https://phoenixnap.com/blog/nagios-monitoring-tutorial).
* Upgrading the app to the newer version.
* Adding new features to the software.

Whenever a user reports a bug or the team discovers a new flaw, the product moves back through its SDLC as many steps as necessary. Some severe defects require updates in the design stage, while most problems take the app back to the development stage.

**The output of this step:** A fully monitored product that continuously sees improvements.

Other less common SDLC phases that are still worth knowing are dedicated steps for deconstructing apps, retiring software, and writing documentation.

**SDLC Methodologies**

Different SDLC methodologies (or models) prioritize different aspects of product creation and measure success in unique ways. Let's look at the most popular SDLC methodologies you can adopt at your company.

**Waterfall**

The waterfall is the oldest and the most straightforward SDLC model on the market. This strategy (also known as **the linear sequential model**) takes a product through a one-way funnel with the following steps:

* Requirement gathering and analysis.
* System design.
* Coding.
* Testing.
* Product deployment.
* Software maintenance.

Waterfall phases run sequentially, and every stage depends directly on the outcome of the previous phase (i.e., each step "waterfalls" into the next one). In a true waterfall model, the team never goes back a step after finishing a phase, so the model's success rests on the team's ability to avoid mistakes.

**Pros of this model:**

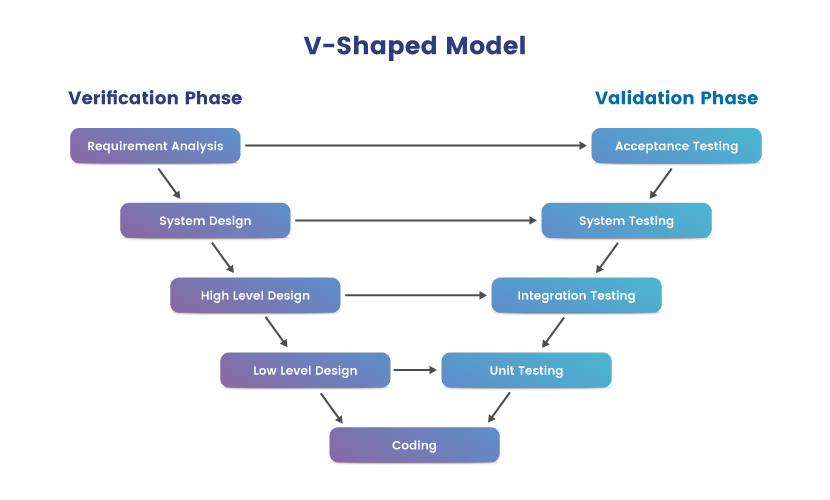
* A tried-and-tested methodology with a logical progression of steps perfect for simple products.
* The model analyzes feasibility at each phase, which helps eliminate bottlenecks and data silos.
* There's a tangible output at the end of each stage.
* A simple SDLC that everyone in the company easily understands.

**Cons of this model:**

* Highly rigid and not a good fit with modern [DevOps teams](https://phoenixnap.com/blog/devops-roles-responsibilities" \t "_blank) that require more flexibility and code revisions after each step.
* Not a good option when working on projects with short notice as there's no way to skip any steps.
* There is little room for change once a step ends (at least not without affecting the cost and delivery time).
* Not ideal for projects with uncertain or changing needs.
* Long build cycles.

**V-Shaped Model**

The V-shaped model (also known as the **Verification and Validation Model**) requires the team to run coding and testing tasks in parallel.



Each verification phase has an assigned validation stage, which makes the model's diagram appear as the letter V (the base of the V is the coding phase). The verification phase consists of the following steps:

* Requirement analysis.
* System design.
* High-level design (the architecture and functionality of modules).
* Low-level design (the architecture and functionality of individual components).

The validation stage has the following steps:

* Unit testing (parallel with the low-level design phase).
* Integration testing (parallel with the high-level design phase).
* System testing (parallel with the system design phase).
* Acceptance testing (parallel with the requirement analysis phase).

A true V-shaped model does not have a dedicated testing phase since each development stage has its own QA sequence.

**Pros of this model:**

* An emphasis on software testing.
* A simple and easily understandable SDLC methodology.
* An extension of the waterfall approach with superior defect detection.
* An excellent fit for small projects with clear requirements that do not change over time.

**Cons of this model:**

* Similar inflexibility issues as the waterfall approach.
* The model does not deal with ongoing software projects as there's no built-in maintenance phase.
* Later changes to the requirements often lead to spikes in costs.

**Prototype Model**

The prototype model requires the team to create a working product prototype during the traditional design phase. When compared to the end-result software, a prototype has the following traits:

* Limited ("dummy") functionality.
* Inefficient performance.

Companies opt for this model to get valuable early feedback from customers. Users offer input on the prototype, developers implement the requested changes, and the team creates a better version of the prototype.

This process goes on until customers have no more negative feedback, after which the team gets a customer-driven requirements analysis and starts developing the final product.

**Pros of this model:**

* Ensures an early proof of concept and validates a genuine need for the product.
* Minimizes the number of defects that reach later phases of the SDLC
* Reliably discovers missing features or functionality.
* Easily adjusts to changes in requirements.

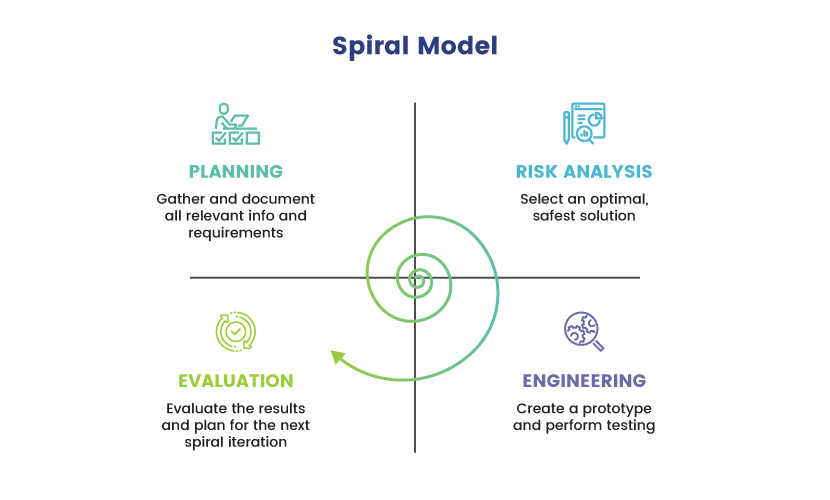
**Cons of this model:**

* Only acts as an SDLC setup and not an entire life cycle.
* More time-consuming than other SDLC methodologies.
* Expensive for advanced apps and products (even impossible for some).

**Spiral Model**

The spiral model is a risk-driven SDLC strategy. This model emphasizes the repetition of its four core stages:

* Determine objectives.
* Identify and resolve risks.
* Development and testing (typically involves some form of prototyping).
* Evaluate results and plan the next iteration.



The idea is to go through these stages repeatedly while making gradual improvements at each pass. The team continuously analyzes needs and makes prototypes while going through the same four-step cycle.

**Pros of the model:**

* The team easily goes back a step in the cycle in case of an error.
* In-depth risk analysis makes this SDLC the go-to model for businesses in [compliance](https://phoenixnap.com/blog/security-vs-compliance)-heavy industries.
* An excellent fit for large projects with hard-to-define requirements and scope.
* Enables teams to adapt to user expectations quickly.
* Little risk of impacting the UX negatively.

**Cons of the model:**

* The cost often spirals out of control if the team goes through too many iterations.
* Requires a skilled team to evaluate when to end iterations and move on to the next SDLC phase.
* An "overkill" for smaller projects with few dependencies and simple requirements.

**Iterative Incremental Model**

The iterative incremental model requires the team to quickly deploy an incomplete version of the software at the end of each development cycle.

The first iteration is a far-from-perfect product that meets a small set of software requirements, and every subsequently deployed version expands the product with more functionality. Each iteration goes through the following phases:

* Inception phase (analysis of the project's needs, goals, and scope).
* Elaboration phase (the design of a functional product architecture).
* Construction phase (coding of the architecture and creation of a deployable product)
* Transition phase (releasing the product to a production environment).

Each iteration goes through verification and requires either user or stakeholder feedback. The last iteration deploys a product version that went through rigorous testing and meets all the requirements specified in the DDS.

Unlike the spiral SDLC methodology (which is similar in concept), the iterative incremental model deploys each software version to production.

**Pros of this model:**

* Easily handles small-to-medium changes in product requirements.
* Gathers user and stakeholder feedback at every iteration.
* Regular risk analysis ensures the product is secure by design and you discover defects early in the SDLC.
* Dividing projects into smaller chunks makes managing the product and analyzing progress easier.

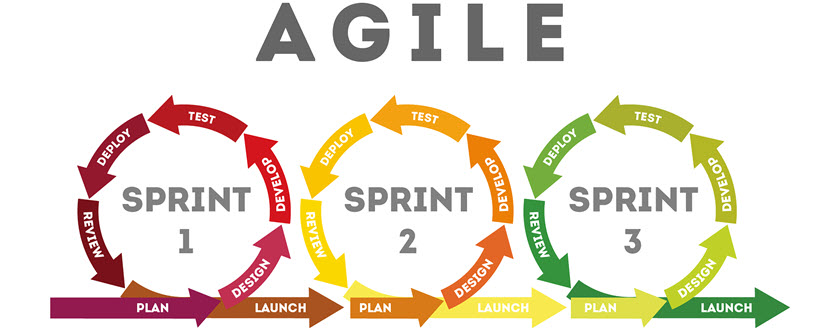
**Cons of this model:**

* Requires a solid understanding of product requirements before you deploy the first iteration.
* The model puts a burden on users and stakeholders by asking for continuous feedback.
* Eats up resources fast if left unchecked.

**Agile Model**

The agile methodology relies on ongoing release cycles that make small, incremental changes to the previous release. Builds evolve as teams add new features and improvements with each deployment.

The agile model requires the team to work in **sprints** that last for 2 to 4 weeks, each with unique requirements and goals. At the end of a sprint, the product owner verifies the code and greenlights its deployment to users. The team then gathers feedback and starts preparing for the next sprint.



Unlike the iterative incremental model, an agile SDLC does not rush the team to deploy the product to customers. Instead, the emphasis is on finding the balance between quality and speed.

The agile approach requires the team to perform testing at the end of each sprint to ensure no potential exploits end up in production.

**Pros of this model:**

* The go-to model for companies wishing to keep up with rapidly changing markets.
* A hand-in-glove fit with [DevOps principles](https://phoenixnap.com/blog/devops-principles" \t "_blank).
* Emphasizes code quality from the start of an SDLC (unlike the iterative incremental or the prototype model).
* Identifies and addresses issues before they evolve into significant problems.
* Makes it easy to get meaningful feedback from stakeholders and end-users.
* A strong emphasis on regular testing boosts [cybersecurity](https://phoenixnap.com/blog/what-is-cyber-security" \t "_blank).

**Cons of this model:**

* Requires an experienced and highly skilled team.
* Maintaining documentation in a fast-paced agile SDLC is a challenge.
* Intense sprints wear a team down over time.

**Big Bang Model**

The big bang model is a high-risk SDLC type that throws most of its resources at development without requiring an in-depth analysis at the start of the cycle.

A big bang starts with little planning and quickly moves to the coding stage. In many cases, developers are the only ones responsible for figuring out requirements, writing code, and checking the validity of a finished product.

**Pros of this model:**

* A high-risk-high-reward SDLC tactic that does not invest much time or money into a project.
* Requires very little planning.
* A good option for simple, low-value products that do not interact with customers.
* A natural choice for small teams and companies without strict formal processes.
* Grants developers the freedom to work on the product in their own way.

**Cons of this model:**

* The lack of upfront planning makes big bangs extremely prone to errors.
* Does not include any built-in testing phases.
* Not a good option for large, ongoing, or complex projects.

Remember that you do not have to pick only one of the models discussed above—many companies benefit from combining two or more SDLC methodologies into a unique hybrid model that fits their specific use case.

**Benefits of SDLC**

A well-defined software development life cycle that fits business needs leads to various benefits, including:

* Lower development costs.
* Improved quality of software products.
* More visibility over the activities of the development team.
* Quicker time-to-market due to better organization, more transparency, and fewer after-the-fact fixes.
* More precise project planning, budget estimations, and scheduling.
* Improved communication between different teams and upper management.
* An improved UX due to fewer bugs and errors reaching production.
* Lower chance of successful [cyber attacks](https://phoenixnap.com/blog/types-of-cyber-attacks" \t "_blank).
* Less chance of project failure.
* Improved quality and exactness of documentation.
* An in-depth understanding of customer and business needs.
* More opportunities for stakeholders to give their input on projects (especially crucial in the early stages of product development).
* A better understanding of the team's current abilities and areas for potential improvements.
* An improved employee retention rate as developers typically like working on well-oiled, SDLC-based projects.
* Less chance of suffering [data corruption](https://phoenixnap.com/blog/data-corruption) or damaging its [integrity](https://phoenixnap.com/blog/what-data-integrity).
* A team culture that emphasizes knowledge sharing and continuous learning.
* Improved service [availability](https://phoenixnap.com/blog/what-is-high-availability) and less chance of unplanned downtime.
* Better-planned-out [backup and disaster recovery (BDR)](https://phoenixnap.com/blog/backup-and-disaster-recovery).

Adopting an SDLC strategy also lowers your team's [technical debt](https://phoenixnap.com/blog/technical-debt) since developers take little to no shortcuts during software creation.



**SDLC Best Practices**

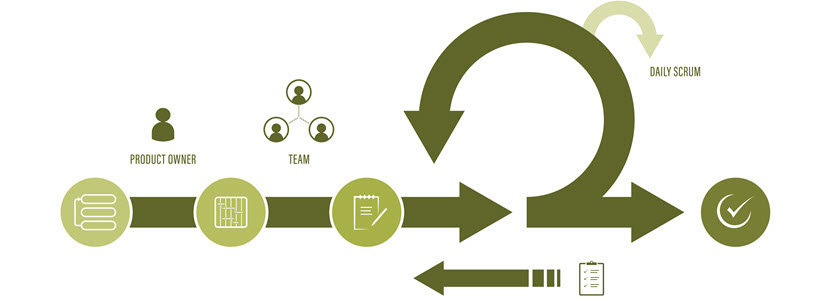
Below is a look at a few best practices you should consider when adopting SDCL at your company:

* **Early-stage security emphasis**: Consider security early in the SDLC, ideally even before the design phase. Remember that [threat modeling](https://phoenixnap.com/blog/threat-modeling) is possible even during the initial analysis phase.
* **Standardize the code review process:** Regular code reviews ensure the team adheres to coding guidelines and prevent bugs from slowing down later SDLC phases.
* **Maintain data hygiene:** Keep all data secure and clean throughout the SDLC. Whether you're dealing with customer, engineering, or market data, the team must maintain hygiene to keep info safe and dependable.
* **Rely on source control:** Using source control ensures the team keeps all code in a single place, which improves security and prevents costly setbacks.
* **Keep up with the current threats:** Ensure your SDLC strategy keeps up with the latest security dangers and best practices. Companies must clearly understand the current threat landscape and maintain up-to-date risk analysis.
* **Rely on automatic tests:** [Automation](https://phoenixnap.com/blog/orchestration-vs-automation) is the best way to ensure tests run regularly and that the team never skips them for expediency.
* **Understand the value of documentation:** SDLC documentation is the go-to source of info for developers, QA specialists, and analysts. These files are the beating heart of an SDLC, so make sure they are accurate and up to date.
* **Organize workshops**: Educate your team on best coding practices, SDCL tools, and frameworks by regularly organizing in-house knowledge-sharing workshops. These events are also ideal for brainstorming sessions that analyze how the team could improve the current SDLC.

Read about [DevOps security best practices](https://phoenixnap.com/blog/devops-security-best-practices" \t "_blank) to see what else companies are doing to boost the safety of their SDLCs and pipelines.

**SDLC Frequently Asked Questions**

Looking for a few more quick takeaways? Let's answer the most frequently asked questions concerning SDLC.



**Why is SDLC Important?**

An SDLC outlines a detailed, step-by-step plan for software development. The practice speeds up decision-making during product creation and minimizes risks while keeping all teams (and stakeholders) on the same page.

**What Is the Secure SDLC (Software Development Life Cycle)?**

Secure SDLC (or SSDLC) is a software life cycle with fully integrated security checks at every phase. Teams start thinking about security risks and measures during the first stage of the SDLC.

**SDLC vs. Agile**

An SDLC is a conceptual outline of the software creation process, while Agile is a project management methodology that focuses on a cyclical, iterative progression while building software.

**SDLC vs. DevOps**

DevOps is a set of practices and philosophies that combines software development and IT operations. This practice takes SDLC concepts to the next level by introducing high levels of automation and focusing on smaller software releases.

**SDLC vs. STLC**

An SDLC defines a sequence of activities during software creation, whereas STLC (Software Testing Life Cycle) refers to a step-by-step list of activities required for reliable software testing.

**SDLC vs. CI/CD**

CI/CD (Continuous Integration and Continuous Delivery) is a set of practices and methods that speed up software delivery by introducing automation into an SDLC. CI/CD is an enhancement to your SDLC, not a potential replacement.

Refer to our comparison of [continuous delivery, deployment, and integration](https://phoenixnap.com/blog/continuous-delivery-vs-deployment-vs-integration) for an in-depth breakdown of the unique roles these practice play in an SDLC.

**The De-Facto Method of Creating Modern Software**

Companies are under more pressure than ever to deliver high-quality products in often unrealistically short time frames. In such a market, approaching software creation in any way that does not involve a pre-defined SDLC is not a business-wise move. Start building your development around a strict life cycle and stay a step ahead of competitors still relying on an old-school approach to product design.

[Share onTwitter](https://twitter.com/intent/tweet?text=What%20is%20SDLC%3F%20Understand%20the%20Software%20Development%20Life%20Cycle&url=https%3A%2F%2Fphoenixnap.com%2Fblog%2Fsoftware-development-life-cycle)[Share onFacebook](https://www.facebook.com/sharer/sharer.php?u=https%3A%2F%2Fphoenixnap.com%2Fblog%2Fsoftware-development-life-cycle)[Share onLinkedIn](https://www.linkedin.com/shareArticle?mini=1&url=https%3A%2F%2Fphoenixnap.com%2Fblog%2Fsoftware-development-life-cycle&title=What%20is%20SDLC%3F%20Understand%20the%20Software%20Development%20Life%20Cycle&source=https%3A%2F%2Fphoenixnap.com%2Fblog)[Share onEmail](mailto:?body=I%20read%20this%20post%20and%20wanted%20to%20share%20it%20with%20you.%20Here%27s%20the%20link%3A%20https%3A%2F%2Fphoenixnap.com%2Fblog%2Fsoftware-development-life-cycle&subject=A%20post%20worth%20sharing%3A%20What%20is%20SDLC%3F%20Understand%20the%20Software%20Development%20Life%20Cycle)

* 

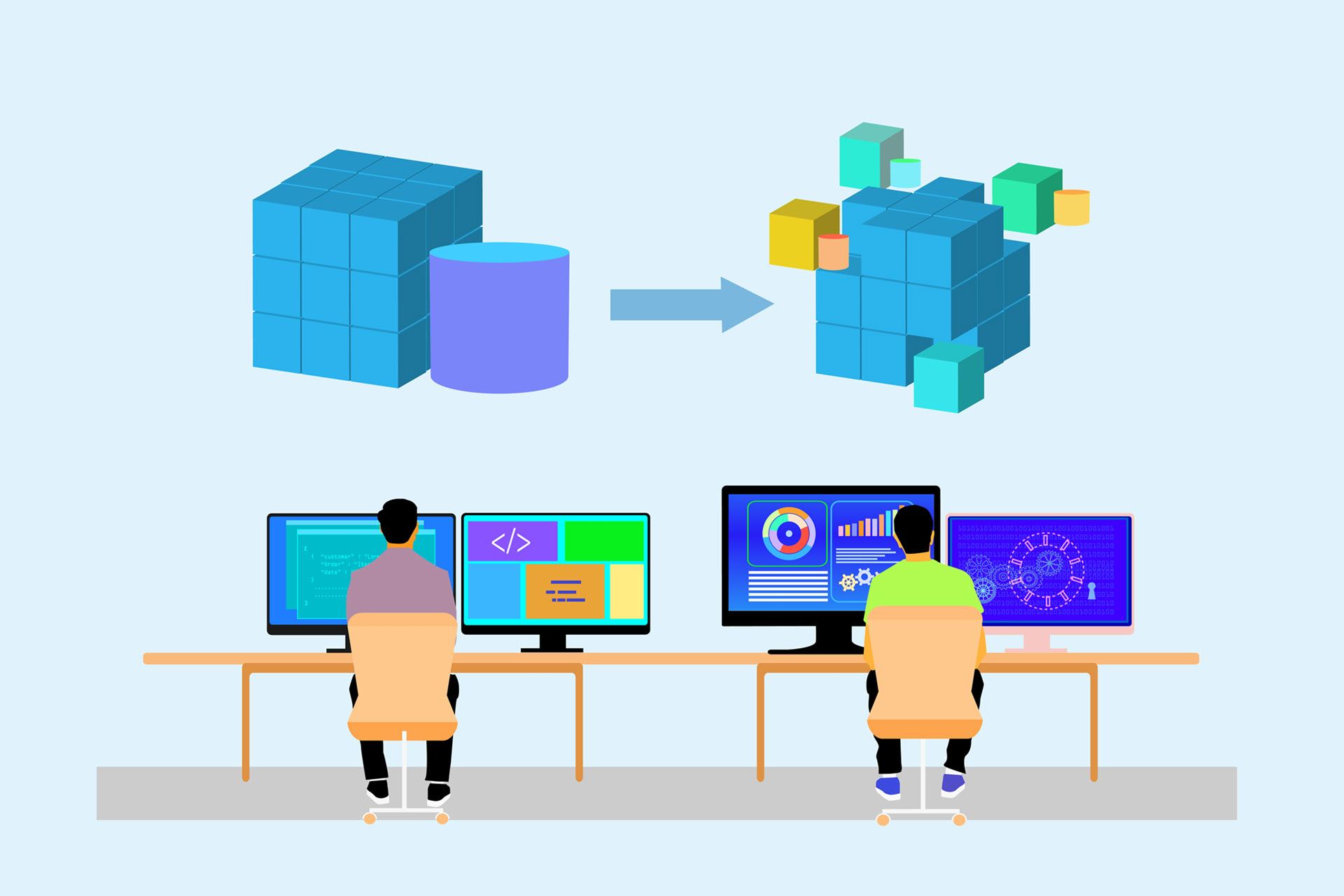
[**What is SecOps? A Security-First Approach to Software Development**](https://phoenixnap.com/blog/what-is-secops)[Security Strategy](https://phoenixnap.com/blog/category/security-strategy)April 7, 2021

Andreja Velimirovic

* 

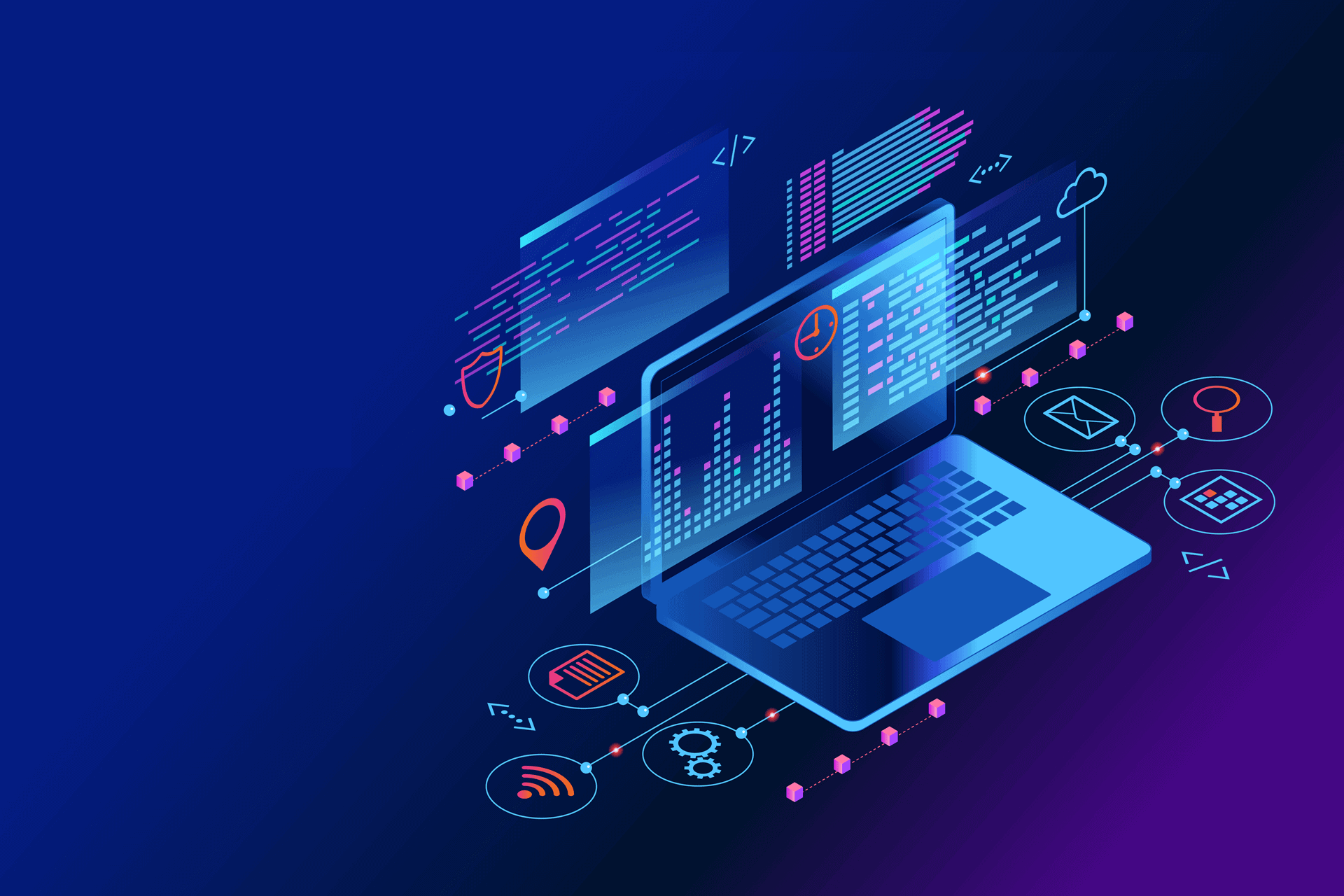
[**15 Best DevOps Tools IT Experts Use for Speed and Agility**](https://phoenixnap.com/blog/best-devops-tools)[DevOps](https://phoenixnap.com/blog/category/devops)October 24, 2022

Andreja Velimirovic

* 

[**What is Container Orchestration? Benefits & How It Works**](https://phoenixnap.com/blog/what-is-container-orchestration)[DevOps,](https://phoenixnap.com/blog/category/devops)[IT Strategy](https://phoenixnap.com/blog/category/it-strategy)September 21, 2022

Andreja Velimirovic

* 

[**DevOps and Virtualization: The Effect of VMs on Software Development**](https://phoenixnap.com/blog/devops-virtualization)[DevOps,](https://phoenixnap.com/blog/category/devops)[Virtualization](https://phoenixnap.com/blog/category/virtualization)March 17, 2021

Andreja Velimirovic



Andreja Velimirovic

Andreja is a content specialist with over half a decade of experience in putting pen to digital paper. Fueled by a passion for cutting-edge IT, he found a home at phoenixNAP where he gets to dissect complex tech topics and break them down into practical, easy-to-digest articles.