codurance | CRAFT AT HEART

Elite Performance

Achieving Speed and Stability with CI/CD and DevSecOps

Ed Farrow
Principal Software Craftsperson

Fane Fonseka
Software Craftsperson

Sam Griffiths
Principal Platform Engineer

2025

codurance.com

Table of Contents

Foreword

The software development industry is riddled with waste and inefficiency! Too many systems are unreliable. It doesn't have to be this way. A handful of key disciplines, applied together, can create a holistic approach that builds quality and speed into the process and eliminates waste.

One of the most recent and valuable additions to these disciplines is the adoption of DORA metrics, the result of some of the most comprehensive research our industry has produced. Used for process improvement, they can deliver significant, measurable gains. Used carelessly, they risk becoming a management stick rather than a tool for genuine improvement.

Equally important are Test-Driven Development and Continuous Integration, practices drawn from Extreme Programming. Combined with Agile Testing at multiple levels and types of automation, these practices cut the risk of failure while tightening delivery cycles.

In commercial environments, techniques from Continuous Delivery and DevOps/DevSecOps are essential. Security, performance, and reliability must be integral to the software we produce. Not afterthoughts bolted on at the end of the lifecycle.

Ed, Fane, and Sam have done an excellent job of distilling these concepts into a compact guide. For the experienced, it is a sharp reminder; for the curious, a clear introduction. I found it valuable to see these practices explained together, their connections made explicit, and their collective impact shown as part of a coherent, holistic approach.

Mashooq Badar
Co-Founder and Software Craftsman at Codurance

Introduction

A core challenge faced by many organisations is the perceived trade-off between the speed of delivery and the stability and security of their systems.

However, the very practices that enable speed (working in small batches, implementing comprehensive automation etc.) are the same practices that create stability, resilience, and security. This creates a positive feedback loop where teams can move faster because they are more stable and secure. This guide provides a proven blueprint for building this capability, transforming your delivery process from a source of friction into a competitive advantage.

Our journey begins with establishing a North Star to serve as a focal point for achieving Elite Performance. From there, we lay the Foundations of CI/CD, build the Engine for developer velocity, and instill the Discipline of XP practices. Finally, we'll provide the Blueprint for a high-performance pipeline, protected by the Shield of modern DevSecOps, and finish by showing the Proof of how these practices drive elite performance.

The path to high performance is not about a single tool or a quick fix; it requires a holistic transformation of process, technical discipline, and culture. This is the journey Codurance guides clients along, achieving transformative outcomes like moving from risky, lengthy release cycles to multiple, reliable deployments per day.

The North Star

Measure what matters

In software delivery, the industry-standard framework for software engineering performance is the DevOps Research and Assessment (DORA) metrics[1].

For over 10 years, the DORA research program has collected and analysed data from thousands of organisations worldwide, identifying the key metrics that consistently correlate with high software delivery and organisational performance. These metrics provide an objective, evidence-based way to measure performance, diagnose systemic issues, and guide investment in technical and process improvements.

The power of the DORA framework lies in its balance between two fundamental pillars: velocity and stability. It recognises that delivering value quickly is meaningless if the product is unreliable, and a stable product that never improves is of little use to customers.

DORA metrics encourage a holistic and sustainable improvement process while avoiding optimising for speed at the expense of quality, or vice versa.

Velocity Pillar

Velocity metrics measure the speed and efficiency of the software delivery process. They provide insight into the organisation's ability to respond to market needs and deliver value to users.

Deployment Frequency

This metric measures how often an organisation successfully releases code to production. It is a direct indicator of throughput and the ability to work in small, manageable batches. High-performing teams can deliver value incrementally and efficiently, leading to faster feedback loops and reduced deployment risk.

Elite performers achieve on-demand deployments, often multiple times per day, whereas low-performing teams may deploy less than once every six months.

Lead Time for Changes

This metric measures the amount of time it takes for a code commit to be successfully running in production. It encompasses the entire delivery pipeline, including code review, automated testing, and deployment processes.
A shorter lead time indicates a highly efficient and automated pipeline, enabling the organisation to react swiftly to customer feedback and changing market conditions.

Elite teams often have a lead time of less than one hour, a stark contrast to the weeks or months taken by lower-performing teams.

[1] Accelerate State of DevOps 10 v2024.3. (2024). Available at: https://services.google.com/fh/files/misc/2024_final_dora_report.pdf.

Stability Pillar

Stability metrics measure the quality and reliability of the software being delivered. They provide insight into the effectiveness of the organisation's quality assurance processes and the resilience of its production systems.

Change Failure Rate (CFR)

This metric represents the percentage of deployments to production that result in a degraded service and require remediation, such as a hotfix or a rollback. It is a crucial indicator of the quality of the development and testing processes. A high CFR suggests potential issues with code review practices, insufficient automated testing, or flawed deployment procedures.

Elite teams maintain a CFR between 0–15%, demonstrating their ability to deliver changes reliably.

Mean Time to Restore (MTTR)

Also known as Mean Time to Recovery, this metric measures the average time it takes to restore service after a production failure. It is the ultimate test of a system's resilience and the team's ability to respond effectively to incidents. This includes any issue that impacts end-users, from a full system outage to severe performance degradation. A low MTTR indicates robust monitoring, effective incident response playbooks, and the ability to deploy fixes quickly.

Elite teams can often restore service in less than an hour, while low performers might take days or weeks.

System-level Diagnostics

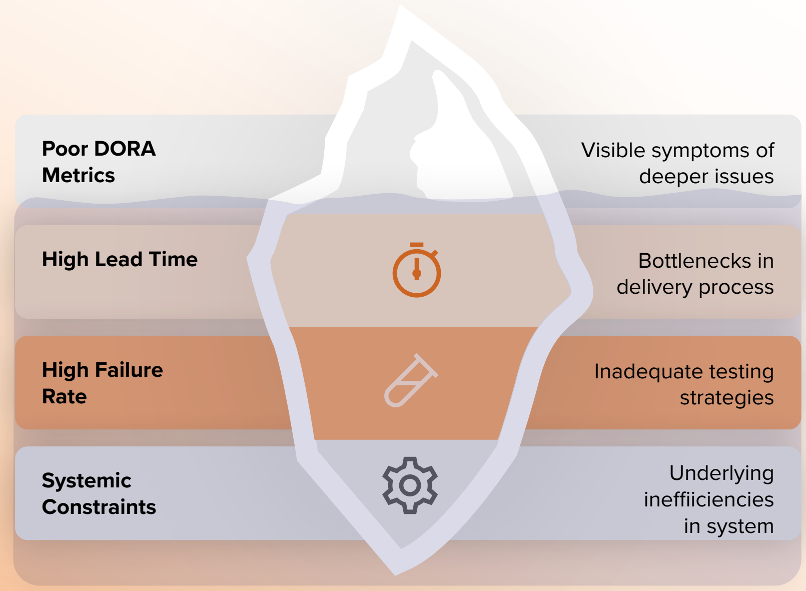
Simply tracking DORA metrics is not the end goal. Their true value lies in their function as a diagnostic tool for the entire software delivery system. A poor metric should not be seen as a failure of a team, but as a symptom pointing to a deeper, systemic constraint or inefficiency.

A **high Lead Time for Change** is a clear signal to investigate the entire delivery process for bottlenecks. This is precisely the kind of systemic friction that the Codurance Value Stream and Delivery Mapping processes are designed to uncover and resolve, helping teams pinpoint and eliminate delays. Improving this metric doesn't just reduce time-to-market; it directly translates into higher developer productivity by shifting effort from reactive firefighting to proactive feature development.

Similarly, a **high Change Failure Rate** is a direct indictment of the quality gates within the pipeline. It suggests that the existing testing strategies are not comprehensive enough to catch defects before they reach production. As well as reducing time to market, improving on this metric also minimises the time spent by developers fixing issues in production and more time on developing new features.

This perspective reframes the objective from “we need to improve our DORA scores” to “we need to use our DORA scores to identify and fix the root causes of issues in our delivery process.” By focusing on improving the underlying practices, the metrics will improve as a natural and sustainable consequence.

DORA metrics are diagnostic tools for software delivery.


Poor DORA Metrics
Visible symptoms of deeper issues
High Lead Time
Bottlenecks in delivery process
High Failure Rate
Inadequate testing strategies
Systemic Constraints
Underlying ineffiiciencies in system

The Foundations

Principles of CI/CD

The Foundations

At the heart of high-performance software delivery are three interconnected practices: **Continuous Integration, Continuous Delivery**, and **Continuous Deployment**.

Continuous Integration (CI)

CI is a development practice where each code merge automatically triggers a build and a run of an automated test suite. CI is best paired with Trunk-based Development, where developers work in short-lived branches and merge regularly. CI and Trunk-based Development provides rapid feedback, ensures the codebase is always in a working state, and fosters collaboration.
It is possible to achieve CI-like automation with other branching strategies like gitflow, but Trunk-based Development works best.

Continuous Delivery (CD)

Continuous Delivery is a practice where every code change that successfully passes all automated tests is automatically packaged into a production-ready release artifact. This ensures that the software is always in a state where it could be released to production at any time.
The final deployment to the live production environment is typically a business decision, often executed with the press of a button.

Continuous Deployment

This is the ultimate state of pipeline automation and represents the highest level of maturity. Every **validated and tested** change is deployed directly to production without any human intervention. This practice maximises the speed of delivery, allowing new features and bug fixes to reach customers within minutes of being completed.

Organisational Trust

The single distinction between Continuous Delivery and Continuous Deployment is the manual approval gate before a production release. This gate only exists (as a safety net) because the organisation, as a whole, does not yet have sufficient trust in its automated quality and security processes to allow releases without a final manual check.

The journey to achieving Continuous Deployment is not primarily about acquiring new tools or writing more automation scripts, but about building a system so reliable, a test suite so comprehensive, and security checks so robust that **the automated pipeline is demonstrably more trustworthy than a final manual review.**

This trust is not granted; it is earned.

Our approach accelerates this journey by focusing on three pillars to build verifiable trust:

- Test-Driven Development (TDD), which ensure code quality at the unit level;
- comprehensive security practices (DevSecOps), which identify vulnerabilities before they are merged; and
- risk-mitigating development patterns like Trunk-Based Development, which keep the main codebase perpetually stable.

This shift in focus from automation to building a foundation of verifiable trust is what enables organisations to remove the final manual gate and unlock the full potential of their software teams.

Reducing the Cost of Change

Every change to a software system, no matter how small, incurs a "transaction cost."

This cost is the total sum of the effort, time, risk, and cognitive load required to move that change from a developer's local machine into the hands of a user in production. In traditional, manual software development models, this transaction cost is exceptionally high. It includes the cost of manual testing cycles, the coordination overhead for "release days," the time spent resolving complex merge conflicts, and the high risk associated with deploying a large batch of changes at once.

The fundamental purpose of a CI/CD pipeline is to drive this transaction cost to as close to zero as possible.

By automating builds, testing, security scans, and deployments, the friction and risk associated with each change is lowered, and the associated transaction cost is reduced.

When the cost and risk of a deployment are low, there is no incentive to bundle many changes together into a large, infrequent release. This creates a powerful feedback loop:

- automation lowers the cost of change, which encourages smaller batches of work; and
- Smaller batches are easier to test, faster to review, and less risky to deploy, which makes the entire automation process simpler, safer, and more efficient.

The Engine

Driving Developer Velocity

Driving velocity with Trunk-based Development

While CI/CD provides the automated pathway for delivery, the source-control branching model dictates the flow of work into that pathway. Trunk-Based Development is the branching model that is most aligned with the principles of Continuous Integration and is a key technical practice consistently adopted by high-performing teams.

Trunk-based Development is a version control management practice where all developers collaborate on code in a single, shared branch. In this model, long-lived feature branches are avoided. Instead, developers commit their work to very short-lived branches that are merged back into the trunk frequently.

As a rule of thumb, branch lifetimes should be measured in hours rather than days.

This constant integration ensures that the trunk is always up-to-date and reflects the collective work of the entire team, minimising the divergence between individual developer environments and the main codebase.

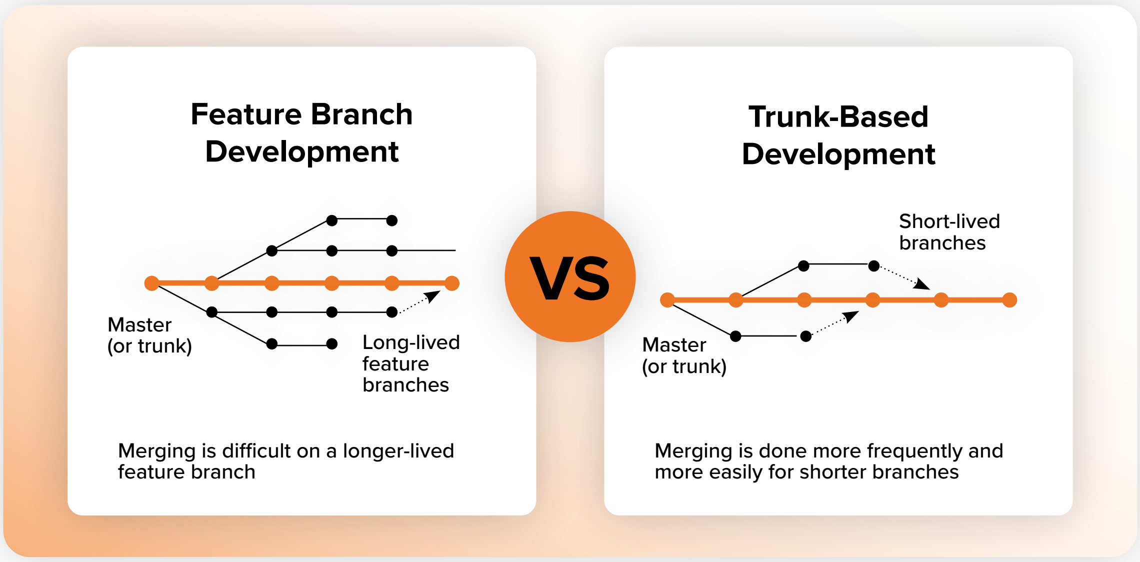
Trunk-based Development vs GitFlow

The choice of branching model is a strategic one that has significant implications for a team's workflow, speed, and ability to practice CI/CD effectively. One of the most common alternatives to Trunk-based Development is GitFlow.

GitFlow is a stricter branching model characterised by multiple, long-lived branches, including main, develop, feature branches, release branches, and hotfix branches. This model was designed for projects with scheduled, versioned releases, where features could be developed in isolation over long periods and then merged together for a release candidate. While this isolation can be beneficial for managing parallel development on large, distinct features, it is fundamentally at odds with the principles of Continuous Integration.

In our work with clients, we consistently observe that teams using GitFlow face significant challenges with delayed integration, resulting in large, complex, and painful merges (the very 'merge hell' that modern practices are designed to prevent).

Trunk-based Development, in contrast, is designed specifically for continuous flow. By enforcing frequent merges into a single trunk, it ensures that integration happens continuously, not as a separate, delayed phase. This minimises merge complexity, provides a constant feedback loop for all developers, and, most importantly, keeps the trunk branch in a perpetually stable and releasable state.


Feature Branch Development
Master (or trunk)
Long-lived feature branches
Merging is difficult on a longer-lived feature branch
VS
Trunk-Based Development
Short-lived branches
Master (or trunk)
Merging is done more frequently and more easily for shorter branches

Aspect	Trunk-Based Development	GitFlow
Branching Model	Simple: A single trunk branch with very short-lived task branches.	Complex: Multiple long-lived branches (main, develop, feature, release, hotfix).
Branch Lifespan	Hours to less than a day.	Days, weeks, or even months.
Integration Frequency	Continuous; at least once per day per developer.	Infrequent; only at the end of a feature's development.
Merge Complexity	Low. Small, frequent merges prevent "merge hell".	High. Long-lived branches diverge significantly, making merges painful and risky.
Code State	Main branch is always stable and releasable.	The develop branch can be in an unstable, intermediate state.
CI/CD Alignment	Excellent. Trunk-based Development is a foundational practice for CI/CD.	Poor. The model inherently creates integration bottlenecks that hinder continuous flow.

Decoupling Deployment and Release

A critical enabling technique for practicing Trunk-based Development safely and effectively is the use of feature flags (also known as feature toggles). A feature flag is a mechanism that allows teams to modify system behavior or turn features on and off in a production environment without deploying new code.

Developers wrap new, incomplete features within a feature flag, ensuring the changes are isolated and dormant unless the flag is turned on. This allows them to merge their unfinished code into the trunk branch safely. The code is deployed to production but remains dormant and invisible to users because the flag is turned off, enabling the business to then decide when to release features without the pressure of a corresponding software release.

Feature flags decouple the technical act of deployment from the business decision to release a feature.

The CI/CD pipeline can run continuously, deploying every change to production, while the business retains precise control over when a new feature is made available to customers. This eliminates the need for long-lived feature branches to hide work-in-progress, enabling true Continuous Integration and Continuous Deployment.

Trunk-based Development: An Architecture Litmus Test

Challenges when adopting Trunk-based Development should be viewed as a powerful diagnostic tool.

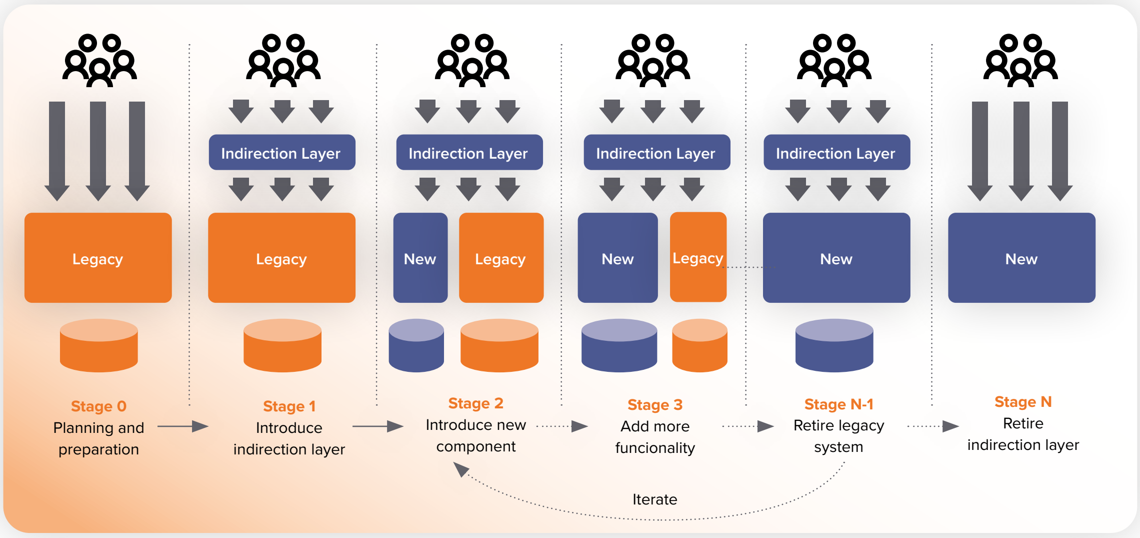
Teams new to Trunk-Based Development sometimes find the practice difficult to adopt. They may discover that nearly every merge to the trunk branch seems to break unrelated parts of the application, causing the build to fail frequently. This is often misinterpreted as a failure of the Trunk-based Development model itself but, in reality, this pain is a key indicator of a flawed application architecture.

In systems with tight coupling, a small change in one part of the codebase can have a large and unpredictable "blast radius," causing cascading failures elsewhere. To make Trunk-based Development sustainable, teams are naturally forced to confront this architectural debt. They must refactor their code towards a more modular, loosely-coupled design, where components are independent, communicate through well-defined contracts or APIs, and can be changed and tested in isolation.

Trunk-based Development reveals underlying architectural weaknesses that are hindering an organisation's ability to move quickly and safely.

Teams should use this feedback to guide their modernisation efforts, leveraging patterns like the Strangler Fig Pattern to gradually transform a tightly coupled system into a modularised and testable platform.

Over time, as the architecture becomes more modular, the friction of practicing Trunk-based Development will decrease, and the team will reap its full benefits of speed and stability.

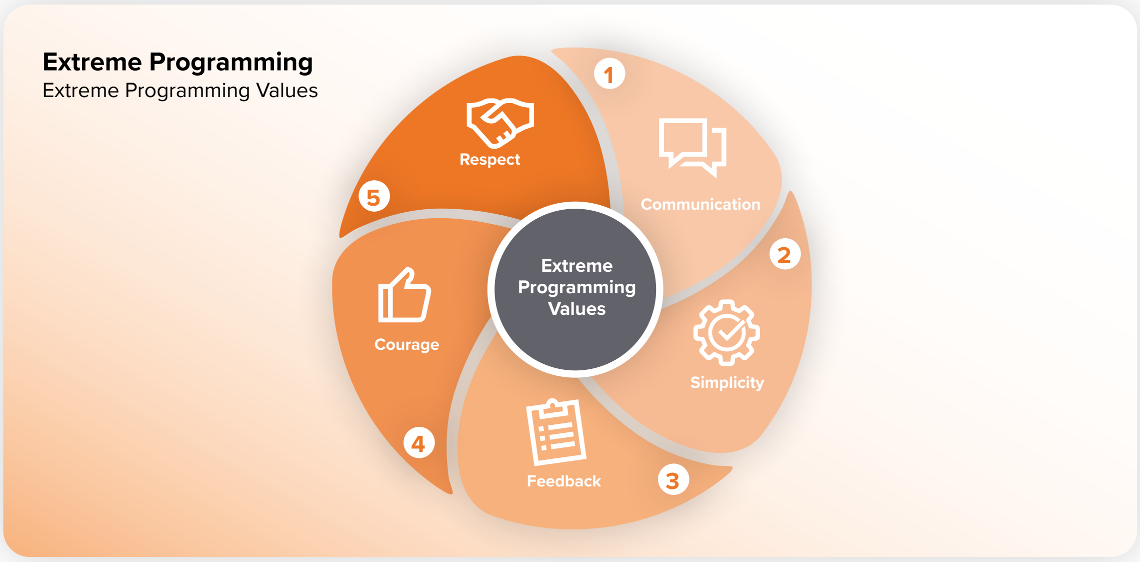


The Discipline

The Role of XP

The modern DevOps and CI/CD practices advocated in this report are not new inventions. They are contemporary implementations of highly disciplined principles that originated with Extreme Programming (XP), an agile software development methodology that has been proven for decades.

XP emphasises technical excellence, rapid feedback loops, and a profound responsiveness to changing customer requirements. The core values of XP (Communication, Simplicity, Feedback, Courage, and Respect) provide the essential cultural foundation upon which these technical practices can thrive.



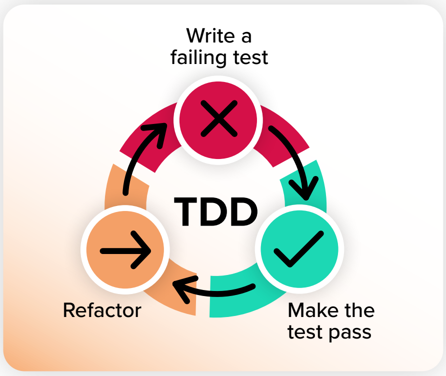
Without this cultural underpinning, attempts to implement CI/CD or Trunk-based Development often fail, as the necessary discipline and collaborative spirit are absent.

The Safety Net

At the heart of XP's technical discipline is Test-Driven Development (TDD). TDD is a software development practice that inverts the traditional coding process. Instead of writing production code and then writing tests for it, TDD follows a short, repetitive cycle known as "Red-Green-Refactor":

- **Red:**
  The developer writes a small, automated test for a new piece of functionality. This test will fail, because the functionality does not yet exist.
- **Green:**
  The developer writes the absolute minimum amount of production code required to make the failing test pass.
- **Refactor:**
  With the safety of a passing test, the developer can now clean up and improve the design of the production code, confident that they are not breaking its behavior.

A comprehensive and, crucially, fast automated test suite is a non-negotiable prerequisite for practicing CI/CD and Trunk-Based Development safely. It is the automated test suite that provides the technical reassurance to merge code frequently into trunk and to refactor the codebase relentlessly to improve its design. This test suite acts as an automated regression-detection system, providing immediate feedback if a new change has inadvertently broken existing functionality.



TDD drives better software

TDD is not a testing practice. TDD is a design practice. It produces a comprehensive test suite as a valuable byproduct, but its primary function is to guide developers toward creating modular, maintainable, and loosely-coupled code.

By forcing the developer to write the test first, TDD shifts their perspective to that of a client consuming the code. Before writing a single line of implementation, the developer must think about how the component will be used, what its public API should look like, and how it should behave under various conditions.

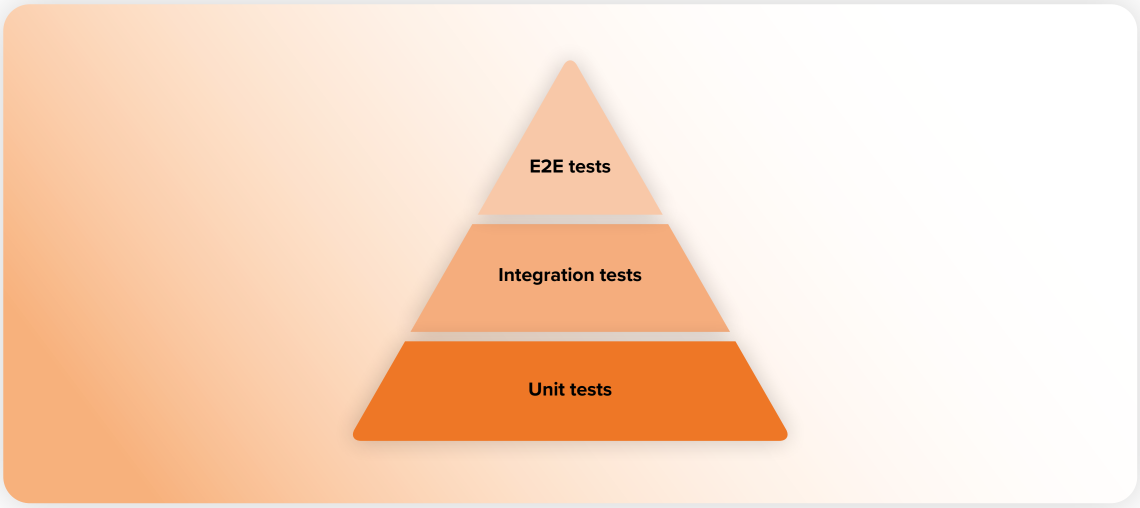
This test-first approach naturally leads to code that is more modular, loosely-coupled, and highly cohesive. TDD makes design feedback immediate and unavoidable, guiding the developer toward creating components that are inherently more maintainable and reusable because they were designed from the outset to be testable.

Anyone on the team can read the tests to understand what a piece of code is supposed to do, and they can run the tests to verify that it still does it.

Furthermore, the resulting collection of unit tests becomes a form of living, executable documentation for the system. Unlike traditional written documentation, which can quickly become outdated and fall out of sync with the actual code, the test suite is an always-current and unambiguous specification of the system's intended behavior.

The Test Pyramid

The Test Pyramid is a strategic model for structuring an automated test suite to achieve the necessary speed while maintaining comprehensive coverage.



Unit Tests (The Foundation)

The vast majority of tests should be unit tests. These tests verify a single unit of code in complete isolation from its dependencies. They must be extremely fast to execute, often running thousands of tests in seconds, and provide precise feedback.

Integration Tests (The Middle Layer)

A smaller number of tests should be integration tests. These slower tests verify the collaboration between two or more components of the system, such as checking if the application code can correctly interact with a database or an external API.

End-to-End (E2E) Tests (The Peak)

At the very top of the pyramid is a very small number of slow E2E tests. These tests automate a user's journey through the application's user interface, validating the entire system from front to back. They can provide a high amount of value through verifying user journeys, but are the slowest, most brittle, and most expensive tests to write and maintain.

A slow build or test cycle is often a symptom of an "Inverted Pyramid", where the team relies too heavily on slow E2E tests.

Teams should strive to cover as much logic as possible with fast unit tests, using integration and E2E tests sparingly to validate the connections between components and check critical user paths.

The Shield

DevSecOps protection

Shifting Security Left

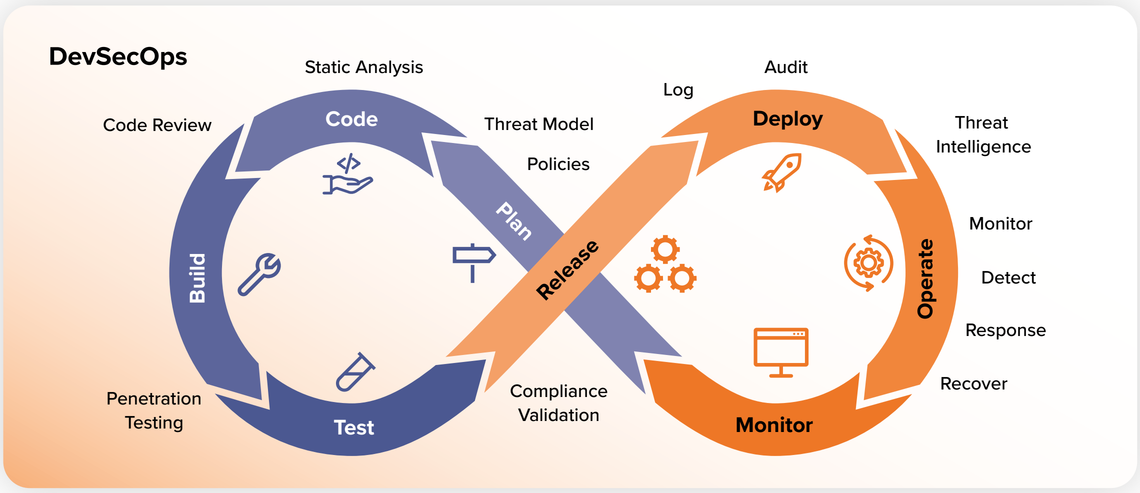
Historically, security has been siloed as a late-stage, specialist-driven phase, handled by a separate team using complex tools, leading to end-of-SDLC audits that delay releases. This siloed approach and its delays are at odds with the rapid, iterative nature of modern CI/CD.

The Three Pillars of DevSecOps represent a fundamental shift in this paradigm. DevSecOps is not a separate team or a final checklist, but a shared responsibility, automated and embedded throughout the entire CI/CD pipeline to ensure that software is secure by design.

DevSecOps is the practice of integrating security testing and culture into every stage of the DevOps lifecycle.

The primary goal of DevSecOps is to "shift security left", finding and fixing vulnerabilities early. This directly improves DORA metrics by preventing security-related change failures and reducing unplanned work, both of which inflate lead times and harm developer morale.

Addressing a security flaw in the code before it is even merged is orders of magnitude cheaper and faster than discovering it in production after a breach has occurred.



Accelerating Delivery

A frequent source of resistance to DevSecOps is the fear that adding security scans will inevitably slow down the pipeline and hinder development velocity. While a poorly implemented, fragmented security program can indeed become a bottleneck, a well-architected DevSecOps strategy is a powerful accelerator.

Production incidents are a major source of disruption, forcing teams into a reactive mode, pulling them away from planned feature development to work on high-priority, unplanned hotfixes.

By integrating fast, automated, and context-aware security checks directly into the developer's workflow, the pipeline catches security defects before they become deeply embedded in the system. This proactive approach prevents defects from escalating into production incidents.

Investing in "shifting security left" is not a tax on development; it is a direct investment in protecting the team's delivery velocity and stability.

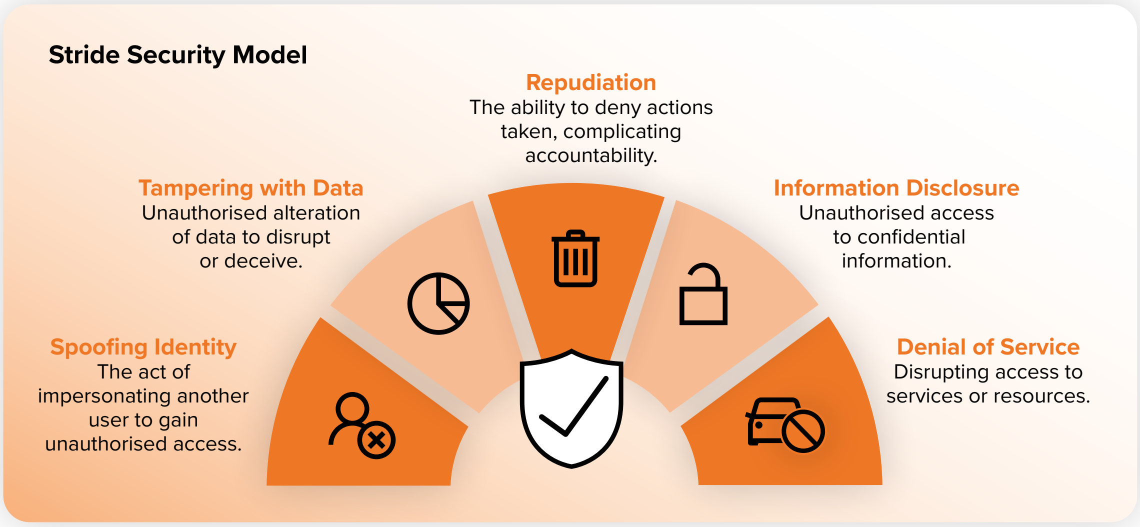
A mature DevSecOps pipeline improves DORA metrics by systematically reducing the volume of security-related failures and the associated rework, allowing teams to ship features faster and with greater confidence.

Pillar 1: Secure by Design

The most effective way to handle security is to prevent security issues from being created in the first place. By integrating security considerations into the earliest stages of the software development lifecycle (SDLC), we can embrace the principle of "shifting left" to its fullest extent, even before a line of code is written.

A cornerstone of this pillar is **Threat Modeling**. This is a structured process where teams proactively brainstorm and analyse potential threats, attack vectors, and vulnerabilities in an application's design. By thinking like an attacker early on, teams can identify and mitigate security risks at the architectural level, which is far more effective and less costly than fixing them later in the development cycle.

Methodologies like STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) provide a framework for systematically identifying a wide range of potential threats.


Stride Security Model
Repudiation
The ability to deny actions taken, complicating accountability.
Tampering with Data
Unauthorised alteration of data to disrupt or deceive.
Information Disclosure
Unauthorised access to confidential information.
Spoofing Identity
The act of impersonating another user to gain unauthorised access.
Denial of Service
Disrupting access to services or resources.

Security should be a foundational component of a system, not a bolt-on.

The output of threat modeling directly informs the security requirements of the system, guiding design decisions and ensuring that security is a foundational component.

Pillar 2: Governance and Guardrails

This pillar focuses on making the secure path the path of least resistance for developers. Instead of acting as restrictive “gates” that block progress, a modern DevSecOps approach establishes automated “guardrails” that guide development teams toward safe practices without hindering their velocity.

There are three core enablers to providing unobtrusive guardrails:

Policy as Code (PaC)

Security and compliance policies are defined in code (e.g., using Open Policy Agent) and integrated directly into the CI/CD pipeline. This allows for the automated enforcement of rules, such as requiring specific security scanners to run or blocking dependencies with critical vulnerabilities. This makes governance auditable, version-controlled, and consistently applied.

Automated Security Checks

The pipeline itself becomes a critical guardrail. By embedding automated security tools directly into the workflow (as detailed in the next pillar), the pipeline provides immediate feedback to developers, allowing them to fix issues within their normal workflow.

Making Security Easy

The goal is to provide developers with tools and training that are seamlessly integrated into their environment. This includes IDE plugins that provide real-time feedback, clear and actionable remediation advice, and reducing the noise of false positives so teams can focus on what matters.

Pillar 3: Detection and Response

While the first two pillars are proactive, this pillar focuses on the detective and reactive controls needed to identify and respond to vulnerabilities that may still emerge.

A robust security posture is not achieved with a single tool. This is where the “defense-in-depth” strategy comes into play, using multiple, layered security scanning tools within the CI/CD pipeline. Each tool category has a specific purpose, ensuring different classes of vulnerabilities are caught at the earliest possible moment. The following slides detail this toolchain.

Detection doesn’t stop when code is deployed. This pillar extends into the operational environment through continuous monitoring. Tools like Security Information and Event Management (SIEM) are critical here, aggregating logs and events from across the entire ecosystem to provide a single pane of glass for security visibility.

This pillar enables teams to respond to incidents quickly and effectively, directly improving the Mean Time to Restore (MTTR) and minimising the impact of any security event on the business and its customers.

The Blueprint

A High Performance Pipeline

The Blueprint

A high-performing pipeline is designed around three core principles:

- failing fast
- building a single immutable artifact, and
- promoting that artifact through progressively more rigorous stages of validation.

A pipeline can be grouped into two main phases:

- pre-merge validation (triggered on a pull request); and
- post-merge deployment (triggered on a merge to trunk).

This section provides a blueprint for a modern CI/CD pipeline based on these principles and phases, focusing on the essential stages and capabilities rather than on specific tools.
The principles and workflow described are universal and can be implemented using any major CI/CD platform, such as GitHub Actions, GitLab CI, Jenkins, or Azure DevOps.

Tooling examples and a Pipeline Maturity Assessment can be found in the Appendices.

Pre-Merge Validation

The entire automated process begins the moment a developer pushes a new commit and opens a pull request (PR) to merge their short-lived task branch into the main trunk branch.

1. **Trigger (On Pull/Merge Request):** This action triggers the pre-merge validation pipeline, which serves as the primary quality gate for the codebase.
2. **Static Analysis:** The very first jobs in the pipeline should provide the quickest possible feedback to the developer. This stage runs directly against the source code without needing to compile or run it, and includes:
    - Linters and Code Formatters to enforce consistent coding style and catch syntax errors.
    - Secrets Scanning to catch inadvertently committed credentials like API keys or passwords.
    - Static Application Security Testing (SAST) to analyse the source code and infrastructure definitions for potential security vulnerabilities and misconfigurations.
3. **Build:** If the static analysis passes, then the app is compiled in a clean, ephemeral, containerised environment to ensure a reproducible build free from any leftover artifacts from previous runs.
4. **Unit Test:** Immediately following the build, the comprehensive suite of unit tests is executed. This is the core quality gate of the pipeline; a failure here indicates a regression in the application's fundamental logic and must block the PR.

Post-Merge Validation

Only after all pre-merge checks have passed, and potentially a human code review has been completed, is the pull request allowed to be merged into the trunk branch, thereby triggering the post-merge process.

5. **Build & Package Artifact:** The first step is to build a single, immutable, and versioned release artifact from the trunk source code. The exact same artifact that is built here will be promoted through all subsequent environments, eliminating the risk of environment-specific build variations.
6. **Automated Acceptance Testing:** The newly created artifact is then automatically deployed to a dedicated, production-like testing environment (often called a staging or QA environment). Once deployed, a suite of automated acceptance tests is executed against the running application, including:
    a. Integration Tests to validate the interactions between the application's components and its external dependencies
    b. End-to-End Tests to validate critical user journeys
    c. Dynamic Application Security Testing (DAST) to find runtime vulnerabilities.
    d. Infrastructure Tests to verify any deployed infrastructure's actual state and behavior.
    e. Software Bill of Materials registration and analysis to track application dependencies

After the artifact has passed all automated acceptance tests, it is ready for production.

Release

At this point we have an artifact, built in a clean environment, which has passed a battery of automated tests to minimise risk. Now, it can be safely released to production.

In a **Continuous Delivery** model, this final step is a manual promotion. An authorised user can trigger the deployment to production with a single click, confident that the artifact has been thoroughly validated.
In a **Continuous Deployment** model, this step is fully automated. A successful acceptance test run will immediately trigger deployment to production.

To further reduce the risk of this final step, teams can adopt advanced deployment strategies including:

- Blue-Green deployments (where a new version is deployed alongside the old one, with traffic switched over instantly); or
- Canary releases (where the new version is gradually rolled out to a small subset of users) can be employed.

The Proof

Driving DORA Improvements

Boosting Velocity

The DORA velocity metrics are a direct reflection of the flow of work through the development system. Our recommended practices are designed to maximise this flow.

Trunk-Based Development and **Continuous Integration** are the primary engines of Deployment Frequency. By requiring developers to merge small, completed pieces of work into trunk at least daily, the system ensures there is always a new increment of value ready to be deployed. This naturally shifts the organisation away from large, infrequent releases toward a continuous stream of small, frequent ones.

A **fully automated CI/CD pipeline** is the single most significant factor in reducing Lead Time for Changes. By eliminating manual handoffs, wait times for testing environments, and manual deployment procedures, a process that once took weeks or months can be transformed into one that can be measured in hours or even minutes.

A fast build and test cycle, enabled by a well-structured Test Pyramid and the TDD discipline, is crucial for a short feedback loop. When developers receive feedback on their changes in minutes, they can iterate quickly, removing a critical bottleneck and further slashing the Lead Time for Changes.

Enhancing Stability

The DORA stability metrics are a reflection of the quality and resilience built into the system. Our recommended practices create a robust safety net that minimises both the likelihood and the impact of failures.

Test-Driven Development and the resulting comprehensive automated test suite provide a powerful regression safety net. With high test coverage, teams have strong confidence that a new change does not break existing functionality. This is the most direct and effective way to lower the Change Failure Rate.

Integrated DevSecOps, with its layered approach to security scanning, prevents entire classes of security vulnerabilities and infrastructure misconfigurations from ever reaching the production environment. By catching these defects early, it further drives down the Change Failure Rate.

When a failure does occur in a system that practices small, frequent deployments, the root cause is <u>almost always</u> one of a very small number of recent changes.
This dramatically simplifies troubleshooting and root cause analysis. Because the change was small, it is often easy to either develop a quick fix and push it through the same fast pipeline or simply revert the change. This ability to rapidly identify and remediate issues is what drives an elite, low Mean Time to Restore (MTTR).

Conclusion

Elite software delivery performance is not a mystery, nor is it the result of chance. It is the direct and predictable outcome of adopting a holistic system of interconnected, disciplined practices. This report has laid out the blueprint for such a system. One that intentionally balances the pursuit of speed with an unwavering commitment to stability and security, creating a sustainable and accelerating pace of innovation.

Achieving this state requires more than just the adoption of new tools. It is a comprehensive transformation that touches on:

- **Process:** Adopting streamlined workflows like Trunk-Based Development and Continuous Integration to enable the smooth flow of value.
- **Technical Discipline:** Committing to rigorous practices like Test-Driven Development that build quality and good design into the code from the very beginning.
- **Culture:** Fostering the XP values of communication, feedback, and courage, and embracing a culture of shared responsibility where security and quality are everyone's job.

This transformation is a journey, and Codurance specialises in guiding organisations through it. Our expertise in software modernisation, platform engineering, and hands-on training in best practices like TDD and CI/CD has a proven track record of delivering measurable results for our clients.

We help organisations not just implement tools, but build the underlying capabilities and culture required for sustained, high-performance software delivery.

Contact us at hello@codurance.com to discuss how we can support your journey to Elite Performance.

Appendices

A Layered Defence

DevSecOps CI/CD Tooling

Category	Purpose	Example Tooling	CI Pipeline Stage
Secrets Scanning	Prevents credentials like API keys, passwords, and private tokens from being committed to the version control system.	Gitleaks TruffleHog Snyk Bearer	Stage 2 (Static Analysis) (Most Important) Stage 5 (Package)
SAST (Static Application Security Testing)	Analyses the application's first-party source code for security vulnerabilities and code quality issues without executing the code.	SonarQube Veracode Horusec Invicti Snyk Checkmarx	Stage 2 (Static Analysis)
SCA (Software Composition Analysis)	Scans the project's dependencies to find known vulnerabilities and license compliance issues in the third-party and open-source libraries being used.	Snyk Dependency-Check Black Duck Trivy Veracode	Stage 2 (Static Analysis) Stage 5 (Package)
IaC Scanning (Infrastructure as Code)	Scans infrastructure definition files for security misconfigurations	Checkov Snyk TFLint Terrascan Trivy	Stage 2 (Static Analysis)
Container Image Scanning	Scans the layers of a built container image for known vulnerabilities in the base operating system and any installed system packages.	Trivy Snyk Grype	Stage 5 (Package)
DAST (Dynamic Application Security Testing)	Tests the *running* application from the outside-in by simulating external attacks. It identifies runtime vulnerabilities and configuration issues that static analysis tools cannot see.	Detectify OWASP ZAP Invicti Snyk Beagle Security	Stage 6 (Acceptance Testing): Runs against the application after it has been deployed to a staging or testing environment. It provides a "black box" assessment of the application's security posture as it would appear to an external attacker.
Infrastructure Testing	Goes beyond static IaC scanning by deploying the infrastructure into a temporary, isolated environment and running tests to validate its actual runtime behavior, configuration, and security posture.	Terratest	Stage 6 (Acceptance Testing): Confirms that the provisioned infrastructure is working as intended (e.g., security groups are correctly applied and blocking traffic, IAM policies are enforced, endpoints are reachable only from expected sources).
SIEM (Security Information and Event Management)	Aggregates and analyses log data from all environments (including the CI/CD pipeline and production) to detect suspicious activity, correlate security events, and facilitate incident response. Provides a holistic view of the security posture.	Splunk Datadog Microsoft Sentinel	Consumes logs and events from all stages, particularly Stage 6 (Acceptance Testing) and Stage 7 (Production), to provide real-time threat detection and post-incident forensic analysis.

Pipeline Maturity Assessment

Assessing organisational CI/CD readiness

Pipeline Maturity Checklist (Pre-Merge)

This checklist, in three sections (Pre-Merge, Post-Merge and Release), provides a consolidated view of the key stages and considerations for building a high-performance CI/CD pipeline, drawing directly from the pre-merge validation, post-merge validation, and release steps detailed in the previous slides. Assess your organisation against each Key Capability to understand where you can continue to improve.

Category	Key Capability	Description	Implementation Status
Triggers	Automated PR/ MR Trigger	The pipeline is automatically triggered on every pull request or merge request to provide immediate feedback.	
Static Analysis	Linting & Formatting	Enforces consistent coding style and catches syntax errors using automated linters and code formatters. Highly language specific.	
Static Analysis	Secrets Scanning	Scans code and commit history for inadvertently committed credentials like API keys or passwords.	
Static Analysis	Static Application Security Testing (SAST)	Analyses source code and infrastructure code for potential security vulnerabilities before the code is compiled or run.	
Build & Unit Test	Reproducible Builds	Code is built in a clean, containerised, and ephemeral environment to ensure consistency and prevent "works on my machine" issues.	
Build & Unit Test	Automated Unit Testing	A comprehensive and fast suite of unit tests is executed to validate the fundamental logic of individual code components.	
Quality Gate	**Pre-Test Quality Gate**	**The pull request or merge request is blocked from merging if any pre-integration test checks (static analysis, tests, etc.) fail.**	
Acceptance Test	Automated Environment Creation	An ephemeral production-like environment is created for further testing.	
Acceptance Test	Targeted Integration Testing	A targeted subset of Automated Integration Tests are executed to validate the interactions between the changed components and their external dependencies.	
Quality Gate	**Post-Test Quality Gate**	**The pull request or merge request is blocked from merging if any integration test fails.**	

Pipeline Maturity Checklist (Post-Merge)

Category	Key Capability	Description	Implementation Status
Triggers	Automated Deployment Trigger	A merge to the trunk branch automatically triggers the deployment pipeline, starting the process of releasing the change.	
Build & Package	Immutable Artifact Creation	A single, immutable, and versioned release artifact (e.g., a container image) is created to ensure consistency across all environments.	
Build & Package	Container Image Scanning	The final container image artifact is scanned for known vulnerabilities in its base operating system and installed packages. The artefact is prevented from progressing if any are present.	
Build & Package	SBOM Generation	A Software Bill of Materials (SBOM) is generated to create a complete inventory of all software components and dependencies.	
Quality Gate	**Pre-Test Quality Gate**	**The release is blocked from progressing if any Build & Package step fails.**	
Acceptance Testing	Automated Staging Deployment	The validated artifact is automatically deployed to an ephemeral production-like staging environment for further testing. Make sure to have automated data seeding and destroy the environment after testing to prevent contamination between tests.	
Acceptance Testing	Automated Integration Testing	Automated integration tests are executed to validate the interactions between the application's components and its external dependencies.	
Acceptance Testing	Automated End-to-End (E2E) Testing	A minimal suite of critical end-to-end (E2E) tests is run to validate key user journeys from start to finish.	
Acceptance Testing	Dynamic Application Security Testing (DAST)	The running application is scanned in the staging environment to find runtime vulnerabilities that are not visible in the source code.	
Acceptance Testing	Infrastructure Testing	Infrastructure tests are run to validate the actual state and behavior of the deployed infrastructure.	
Quality Gate	**Post-Test Quality Gate**	**The release is blocked from progressing if any acceptance test fails.**	

Pipeline Maturity Checklist (Release)

Category	Key Capability	Description	Implementation Status
Production Deployment	Automated Production Deployment	Deployment to nightly, staging/QA and production is automated, either manually triggered (Continuous Delivery) or fully automated (Continuous Deployment).	
Production Deployment	Feature Flag Enablement	Feature flags are used to deploy code to production with new features disabled, allowing the business to control the timing of the release to end-users, perform canary releases, or conduct A/B testing.	
Production Deployment	Feature Flag Environments	Uses feature flags to continuously deploy the latest release to staging/QA/nightly environments. This ensures these environments are never stale and allows incomplete features to be tested by QA teams by enabling the flag, while they remain hidden from production users.	
Production Deployment	Advanced Deployment Strategies	Advanced deployment strategies like Blue-Green or Canary releases are used to reduce deployment risk and allow for safe rollouts.	

codurance

hello@codurance.com
www.codurance.com

Codurance is a global software consultancy that helps businesses build a better, sustainable technical capability that supports growth and innovation.

We build well-crafted, reliable, secure and easy to modify software that minimises waste, and reduces cost and delivery times.

For more information visit: **www.codurance.com**

Follow us on social media:
@codurance