International Journal of Leading Research Publication (IJLRP)

E-ISSN: 2582-8010 ● Website: www.ijlrp.com ● Email: editor@ijlrp.com

Optimizing Delivery Cycles In Salesforce: A Study On Managed Packages And Quick Turnaround Strategies

Uday Kumar Reddy Gangula

ukgangula@gmail.com

Abstract:

The research explores methods to enhance software delivery cycles that operate on the Salesforce platform. The paper delivers an extensive evaluation of how development approaches shifted from traditional org-based methods to modern source-based approaches. The research evaluates First-Generation (1GP) and Second-Generation (2GP) managed packaging alongside legacy deployment tools, Change Sets, the Ant Migration Tool, and the Salesforce DX tool to assess transformative effects. The research combines architectural analysis with established DevOps principles to identify essential strategies for quick turnarounds through patch release tactics, unlocked package adoption, and CI/CD pipeline implementation. The research shows that enterprises can achieve the best results through the complete adoption of source-driven development, which uses 2GP and Salesforce DX to boost development speed and predictability and minimize operational costs.

Index Terms: Salesforce, Managed Packages, Salesforce DX, DevOps, Continuous Integration, Software Development Lifecycle (SDLC), Release Management, 1GP, 2GP

I. INTRODUCTION

A. The Imperative for Agility in Enterprise Application Development

The speed of software delivery stands as the main factor that determines competitive success in modern digital economic systems. Organizations experience continuous demands to innovate and adapt to market changes while delivering exceptional value to customers at an unprecedented speed. Organizations need a software development lifecycle (SDLC) that combines efficiency with repeatability and extreme agility. The need for agility in the Salesforce platform directly affects technical teams because numerous enterprises use this platform to manage their critical business operations, including sales and customer service, marketing, and commerce. Organizations need to shorten their development and delivery cycles to enable quick new feature and enhancement releases without compromising software quality or system stability. [1] This paper addresses the main business challenge by examining the evolution of Salesforce development practices to identify methods that enhance delivery cycle performance.

B. The Salesforce Platform: A Double-Edged Sword of Power and Complexity

The combination of declarative and programmatic tools in Salesforce's metadata-driven multitenant architecture enables fast application development, which drives the platform's success [2]. Large-scale enterprise organizations develop into "happy soup" over time because their interdependent metadata becomes complex while documentation remains minimal [3]. The complexity of the system creates agility challenges because basic changes trigger multiple effects, deployments become unstable, and innovation slows down because the platform acts as a performance barrier [4].

C. Thesis Statement

The process of optimizing Salesforce delivery cycles needs more than process adjustments because it requires a complete transition to source-driven development. Teams can achieve speed, reliability, and scalability through their transition from 1GP and manual methods to 2GP and Salesforce DX and DevOps practices [5]. The success of this approach depends on adopting a "packaging mindset," which includes modularity, version control, automated testing, and CI/CD [4]. Technical adoption without cultural alignment between teams results in no significant improvements [1].

II. THE EVOLUTION OF SALESFORCE PACKAGING MODELS

A. First-Generation (1GP Managed Packages: The Org-Centric Foundation

The first-generation packaging model, known as 1GP, functions as the original method for distributing applications through Salesforce. The packaging org functions as both the source of truth and namespace holder while maintaining tight development and versioning connections to this specific org. The Salesforce UI remains the primary tool for packaging because each org can support only one package and one namespace [5].

Release and Versioning Model:

The 1GP versioning model is linear and rigid. The Beta state of packages enables testing but prevents production installation and does not support upgrades. Once promoted to "Released," packages become immutable—components cannot be deleted or significantly modified (e.g., changing a custom field's data type) [6]

Patch Mechanism:

The bug-fixing process requires developers to establish separate "patch organs" that originate from their main version. The manual and isolated process preserves the org-centric nature of 1GP yet prevents automation capabilities [5].

Architectural Constraints: 1GP architectural constraints impose several limitations:

- Restrictive Code Sharing: The code-sharing process under 1GP proves difficult to manage because it depends on unsafe workarounds, whereas 2GP enables structured code sharing through source control and modular APIs [7].
- Limited Automation: The critical packaging operations need manual UI steps, which restrict the integration of CI/CD pipelines [5].
- High-Friction Releases: The process of manual installation for updates by each customer org administrator creates delays in adoption and slows down value delivery [8].
- Complex Branch Management: The maintenance of multiple versions becomes complex because it demands separate patch lines, which results in elevated maintenance costs and more extended-release periods [8].

B. Second-Generation (2GP Managed Packages: A Source-Driven Paradigm

The Salesforce DX tool suite introduced 2GP, which transforms packaging through source-driven methods. The Salesforce CLI deploys metadata and source code stored in a VCS like Git, which provides traceability and automation capabilities [5].

Dev Hub and Namespace Management

The Dev Hub org serves as a centralized system for managing package creation and lifecycle operations in 2GP. The Dev Hub in 1GP differs from 1GP because it can handle multiple packages through shared namespace management, which supports modular, interoperable package development [5].

1) *Namespace Management:* The management of namespaces becomes more adaptable in 2GP. The namespace creation process begins in a standard Developer Edition org before linking it to the Dev Hub. The main benefit of this approach is that it allows different 2GP packages to utilize the same namespace. [5] Developers can create multiple related packages through this approach, which function together as if they were part of one unified application.
2) *Release and Versioning Model:* The 2GP release and versioning model is designed for automation and

flexibility.

- API and CLI Support: All packaging operations are accessible via CLI, which aligns with CI/CD automation practices [5].
- Flexible Versioning: Developers can branch and version from any prior release, supporting parallel development workflows and reducing risk.
- Streamlined Patching: Patches can be created via CLI commands and deployed through automated pipelines, avoiding the need for separate patch organizations [7].

3) Architectural Advantages: The source-driven architecture of 2GP provides several advantages that directly contribute to faster and more reliable delivery cycles:

- **Modular Code Sharing:** With Second-Generation Managed Packages, developers can share Apex code across multiple packages using the @namespace Accessible annotation on public classes and methods. This enables clean, modular API design between packages that share a namespace, avoiding the use of the global modifier and maintaining tighter control over visibility and limits.
- **Declarative Dependencies**: The sfdx-project.json file outlines package dependencies, enabling the Salesforce CLI to manage these relationships during the creation of package versions effectively. This method enhances architectural transparency, promotes modular development, and aids in automating CI/CD workflows. [5]

C. Comparative Analysis of 1GP and 2GP

The two packaging generations exhibit significant differences that impact development speed, team collaboration, and operational growth.

The 2GP architectural features, including shared namespaces, declarative dependencies, and CLI-driven automation, enhance the packaging process beyond its basic functionality. The necessary components exist to build an advanced microservices-style architecture on the Salesforce platform. The one-to-one relationship between a package and a namespace in 1GP creates such high overhead that it discourages developers from dividing large applications into smaller independent parts. Monolithic package design becomes the only viable option for developers.

The following table provides a comparative summary of their key characteristics.

TABLE I- COMPARATIVE ANALYSIS OF 1GP AND 2GP PACKAGING MODELS

Feature	**First-Generation (1GP) Managed Package**	**Second-Generation (2GP) Managed Package**
Source of Truth	Packaging Org [5]	Version Control System (VCS)
Development Model	Org-based [5]	Source-driven [7]
Core Tooling	Salesforce UI	Salesforce CLI
Package Ownership	Packaging Org	Dev Hub
Namespace Scope	One namespace per package	One namespace shared across multiple packages
Code Sharing	global Apex	@namespaceAccessible public Apex

Versioning	Linear and restrictive	Flexible and branch-friendly [7]
Patching	Requires dedicated Patch Orgs	Simple CLI command [7]
Automation	Partial API coverage, manual steps required	Fully scriptable via API and CLI [5]
Dependency Management	Implicit, managed via UI	Declaration in sfdx-project.json file [5]

The 2GP architecture enables developers to build applications using microservices-style development while also simplifying packaging. The 1GP monolithic structure prevents teams from creating independently deployable packages named "Quoting," "Billing," and "Fulfillment" that operate under a shared namespace with independent release cycles [5]. The modular design supported by VCS enables parallel development and better dependency isolation [7]. The 2GP tool provides efficient packaging capabilities while establishing a base for building scalable enterprise applications on the Salesforce platform.

III. TRADITIONAL DEVELOPMENT AND DEPLOYMENT LIFECYCLES (PRE-SFDX

The Salesforce development lifecycle primarily operated through org-centric methods before Salesforce DX gained widespread adoption and modern DevOps principles became popular. The development model featured extended environment lifecycles, along with manual deployment methods, which made it challenging to achieve both speed and reliability.

A. The Org-Centric Model and Sandbox Strategies

The standard SDLC in the pre-SFDX era was structured around a linear cascade of persistent sandboxes. [3] The typical flow involved:

1) **Development:** Each developer maintained their Developer or Developer Pro Sandboxes to write code and configure settings. [9]
2) **Integration & QA:** The process of merging changes from different developers into a shared Integration or QA sandbox required manual deployment for testing purposes. [3]
3) **User Acceptance Testing (UAT):** The consolidated changes were deployed to a UAT or Staging sandbox (often a Partial or Full Copy sandbox) for business users to validate after QA completion. [3]
4) **Production Deployment**: After receiving UAT sign-off, the changes were deployed to the production org.

The primary and ongoing issue with this model involved environmental drift. The long duration of sandboxes, together with their ability to receive independent modifications, led to configuration differences between production and other sandboxes. [9] A developer would introduce minor manual changes to a QA sandbox for testing purposes but forget to include these modifications in the final deployment package. The practice of deploying changes that worked in testing environments but failed in production occurred frequently because of this drift. [10] The process of refreshing sandboxes from production to match production settings required extensive time and occurred mainly after major releases, which allowed drift to build up throughout time. [11]

B. Legacy Deployment Mechanisms: Change Sets and the Ant Migration Tool

The deployment landscape of this era featured two main tools that served distinct user personas and use cases.

1) *Salesforce Change Sets:* The native UI-based Change Sets tool functions as the native deployment

tool for moving metadata between connected organizations. The process requires manual intervention from administrators who choose specific components for the "Outbound Change Set" before uploading it to the target org (e.g., from sandbox to production) and then validating and deploying the corresponding "Inbound Change Set" in the target org's UI. [12]
The tool gained popularity among administrators and users who avoided command-line tools because it provided an easy-to-use interface. The method presented major drawbacks, which made it inappropriate for handling complex or fast-paced release cycles. [13]

- Manual effort: The manual process required time-consuming selection of components, which often resulted in missing essential elements [1].
- Component gaps: The deployment system lacked support for picklists, sales processes, and other metadata types [1].
- No support for deletions or renaming: The system did not support deletion operations or renaming functions.
- File/path limits: The system-imposed file/path restrictions require users to perform repeated cloning operations across different stages.
- No version control: The system lacked version control capabilities, which made it incompatible with Git-based workflows [13]

Force.com Ant Migration Tool
The Force.com Ant Migration Tool functioned as the standard automation solution for developers and teams. [14] The Java/Ant-based command-line utility functions through interactions with the Salesforce Metadata API. [15] Developers could use this tool to execute scripts that retrieved and deployed metadata from an XML manifest file named package.xml. [16]
The Ant tool offered capabilities far beyond Change Sets:

- **Automation and Scripting:** The tool functioned as an automation system that developers could use to perform scriptable deployments for large-scale and repetitive migration operations. [14]
- **Integration with CI Servers:** The tool operated through command lines, which enabled its integration with Jenkins and other automation servers to create basic CI/CD pipelines. [17]
- **Component Deletion:** The tool allowed deletion of components from target organizations via destructiveChanges.xml files, offering a crucial functionality that Change Sets lacked. [16]

Despite its power, the Ant Migration Tool had its own set of limitations:

- **High Technical Barrier:** The tool required developer-level expertise, encompassing knowledge of Java programming, skills with the Apache Ant build tool, understanding of XML, and proficiency in command-line scripting. [13] The tool remained inaccessible to most administrators because of its technical nature.
- **Complex Setup and Maintenance:** The setup process for the tool involved complex and error-prone steps to install Java and Ant versions and configure environment variables. [16]
- **Local Credential Storage:** The standard configuration requires storing Salesforce usernames and passwords in a local build. Properties file, which violated the security policies of numerous organizations. [13]

C. Comparative Analysis of Traditional Deployment Tools
The two legacy tools created a significant split in the Salesforce ecosystem, which required teams to work in separate processes that reduced their ability to collaborate and work efficiently.

D. Inherent Challenges and Process Bottlenecks
The traditional lifecycle introduced significant process risks:

- Frequent deployment failures: The manual steps and complex scripts in the process resulted in deployment errors [1].
- Overwritten changes: The absence of a single source of truth led to frequent overwrites of changes.

- Poor traceability: The system failed to track changes, approvals, and release content.
- "Deployment Day" chaos: The "Deployment Day" process was both rare and stressful because it involved rushed testing and manual corrections, which often failed [10]

TABLE II
I: COMPARISON OF TRADITIONAL DEPLOYMENT TOOLS (PRE-SFDX)

Attribute	**Salesforce Change Sets**	**Ant Migration Tool**
Primary User	Salesforce Administrator, Business Analyst	Salesforce Developer, Release Engineer [13]
Interface	Declarative UI (Point-and-click)	Command-line (XML & Scripting) [14]
Setup Complexity	Low (Built-in)	High (Requires Java, Ant, local setup) [16]
Automation	Manual process [1]	Highly scriptable, schedulable [18]
Component Deletion	Not supported	Supported via destructiveChanges.xml [16]
Version Control Integration	Poor (Not designed for it) [13]	Possible (but requires manual process) [16]
Ideal Use Case	Small, simple, infrequent deployments [13]	Large, complex, repetitive deployments [14]
Key Limitation	Manual effort, component gaps, no rollback [1]	Technical complexity, security concerns [13]

IV. THE ADVENT OF SALESFORCE DX AND MODERN DEVELOPMENT

The Salesforce Developer Experience (DX) tool suite made its debut at Dreamforce in 2016, which became a defining moment for the platform. [19] The platform transitioned purposefully from its traditional org-based model to adopt a contemporary source-driven development approach that accepted software engineering tools and practices.

A. Core Tenets of Salesforce DX

Salesforce DX introduced a completely new development philosophy that went beyond providing tools for platform development. The fundamental principles of Salesforce DX aim to solve the fundamental problems of the traditional lifecycle.

1) New Philosophy

The fundamental concept of Salesforce DX established that version control systems (VCS) should serve as the definitive source of truth. The organization-centric model is no longer the leading approach because the production or packaging teams do not have sole authority over the application’s final state. Salesforce DX has created a basis for team collaboration and automation by allowing the management of source code and metadata using Git version control systems. [19]

2) *Salesforce CLI:* The Salesforce Command-Line Interface (CLI) represents the central element of the new experience. The CLI functions as a robust tool that unites capabilities from the Ant Migration Tool and adds multiple new features to the system. [20] The CLI established itself as the main scriptable interface that developers used to interact with their organizations, execute metadata manipulation test execution, and automate the application lifecycle. [21]

3) *Scratch Orgs: Ephemeral and Source-Driven Environments:* The introduction of scratch orgs represented a fundamental transformation in Salesforce DX. The Salesforce environment known as scratch org exists as a temporary and customizable platform that developers can establish through project-scratch-def.json files stored in their Version Control System. [20] Developers could establish a blank org through the feature branch source code deployment, followed by development and testing in isolated conditions before discarding the org after completion. [21] The ephemeral environmental approach eliminates the environmental drift issues that affect the traditional long-lived sandbox model. Every developer and automated test run against a predictable environment that directly derives from the source of truth.

4) *The Dev Hub:* Salesforce introduced Dev Hub as a solution to handle the new world of ephemeral organizations and modern packages. The Dev Hub operates as a production org feature, serving as the central control center for managing all scratch orgs and 2GP packages related to projects or teamwork.

B. Enabling Continuous Integration and Continuous Delivery (CI/CD)

The Ant Migration Tool allowed developers to create CI/CD pipelines, but the process remained complex and brittle. Salesforce DX provides all necessary primitives through its CLI-first approach and source-driven model to build reliable and automated CI/CD pipelines as a fundamental development process feature. [11]
A typical CI workflow for a Salesforce project using a modern toolset like Jenkins proceeds as follows [17]:

1) A developer who works on new feature development uses Git repository feature branches to store their code.
2) The commit action sends an automatic webhook notification to a CI server, which includes Jenkins as an example.
3) The Jenkins server starts a predefined build job, which executes a sequence of Salesforce CLI commands through a script.
4) The script performs a series of automated quality gates.
   - The script uses sfdx force:org:create to create a fresh scratch org from the project definition file.
   - The script uses sfdx force:source:push to transfer source code from the developer's feature branch to the scratch org.
   - The script executes all Apex unit tests through sfdx force:apex:test:run to validate logic and verify code coverage standards.
   - The process includes an optional step to execute static code analysis tools for security vulnerability detection, code quality assessment, and formatting standard enforcement.
5) The Jenkins job finishes by sending build results to developers through pull request status checks. A successful build indicates that the new feature has sufficient safety for merging into the main development branch.

The automated feedback system enables teams to detect bugs and integration problems at an early stage, which decreases development costs and reduces defect risks while speeding up overall development time.

Salesforce DX delivered more than improved development tools for professionals. SFDX established a standardized metadata source format while offering a scriptable API to modify it, which enabled the development of a new tool ecosystem. [21] The standardization of metadata source formats through SFDX established a common language and predictable foundation for building new tools. [1] The new opportunity allowed third-party DevOps vendors to build user-friendly graphical interfaces that operate on top of the SFDX architecture. Through these tools, administrators can work declaratively in a sandbox before the system converts their changes into SFDX source format and performs Git branch commits. CI/CD pipeline triggers with a single button click. The essential plumbing provided by SFDX enabled the development of unified release processes, which benefited the entire team regardless of their technical expertise. A mature DevOps culture requires this unification as its fundamental element to optimize delivery cycles.

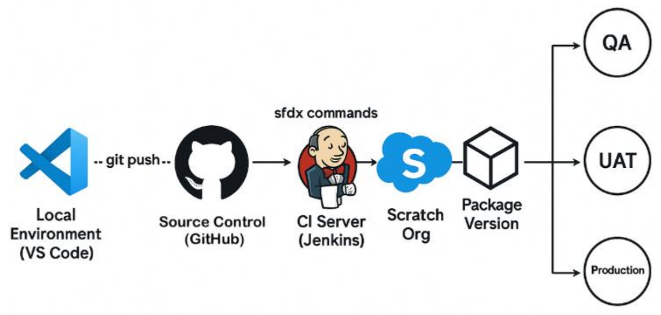

Fig. 1. Automated Salesforce Deployment Pipeline: From Developer Commit to Production Release

V. STRATEGIES FOR OPTIMIZING DELIVERY CYCLES

Development teams, along with architects, can implement these practical strategies to enhance their delivery speed and improve release quality based on established packaging models and development lifecycles.

A. Quick Turnaround Tactics: The Strategic Trade-off of Patch vs Major Releases

ISVs, along with any managed package distribution team, face a vital strategic decision between major releases and patch releases, which affects their turnaround time. [8]

- **Patch Releases for Speed and Bug Fixes:** The quickest approach for delivering changes to customers is through patch releases. These automated deployment mechanisms make patches suitable for urgent bug fix distribution to subscriber organizations. [8] The fast delivery speed of patches comes with limited adaptability. Patches enable developers to modify current components, but they cannot introduce new components or delete existing components. [8]
- **Major Releases for Innovation and New Features:** Major releases serve as the necessary path to implement new features, along with component additions and major architectural modifications. Major releases enable innovation through flexible delivery, yet create significant obstacles to deployment. Major releases do not use automatic deployment methods, so administrators must search for and manually install the update in customer organizations. The manual upgrade process creates a significant delay in new functionality adoption, which ultimately reduces the speed of delivering value to end-users. [8]
- **The Technical Debt Trap:** Teams sometimes use patch mechanisms to add new functional enhancements by adding new logic to current components as an antipattern. The approach appears to be efficient for avoiding significant release limitations yet leads to dangerous technical debt accumulation. Incorporating additional features into components creates complex code that is hard to grasp and maintain, resulting in slower future development. [8]
- **Optimization Strategy:** A well-evolved release management approach demands a disciplined and balanced approach for success. The intended use of patches involves delivering high-priority bug fixes quickly. All new functionalities at any size should be included in the upcoming major release schedules. Teams need to implement refactoring as an active method for managing technical debt. During primary release cycles, teams should evaluate all added functionality to existing components through patches before transferring appropriate elements to new purpose-built components to maintain a clean and healthy codebase. [8]

B. Leveraging Unlocked Packages for Internal Agility and Modularity

ISV distribution uses managed packages as its standard but enterprises developing internal customizations gain optimization power through Unlocked Packages.

- **Breaking the Monolith:** Unlocked Packages function as 2GP packages, which help architects and development teams divide big monolithic production organizations into smaller, independent deployable modules. The main instrument for implementing a "packaging mindset" during internal development is Unlocked Packages. [4]
- **Key Characteristics:** Unlocked Packages inherit all 2GP and SFDX toolchain benefits because they operate from source code and follow version control through Salesforce CLI. The key difference between managed packages and unlocked packages is that managed packages keep their Apex code and objects protected, preventing modifications in the organizations where they are installed. Unlocked packages give teams full access to editable code and objects, providing more flexibility for internal teams that don't require the intellectual property protections offered by managed packages.
- **Enabling Parallel Development:** One of the main benefits of utilizing unlocked packages for organizational modularization is the opportunity for parallel development of components. Enterprises can establish different unlocked packages to handle their "Sales," "Service," and "Finance" customizations separately. The development teams operating on other business units can work on their packages independently while following their release schedules without creating dependencies between them. The separation between these components enables faster organizational change implementation because it minimizes development delays.
- **Org-Dependent Unlocked Packages:** Some complex org metadata cannot be packaged because it depends on production org features, including multi-currency and specific standard object configurations. Org-Dependent Unlocked Packages from Salesforce serve as a solution to this problem. Package dependency validation occurs during installation time instead of package creation time, which enables modularity for complex metadata structures. [4]

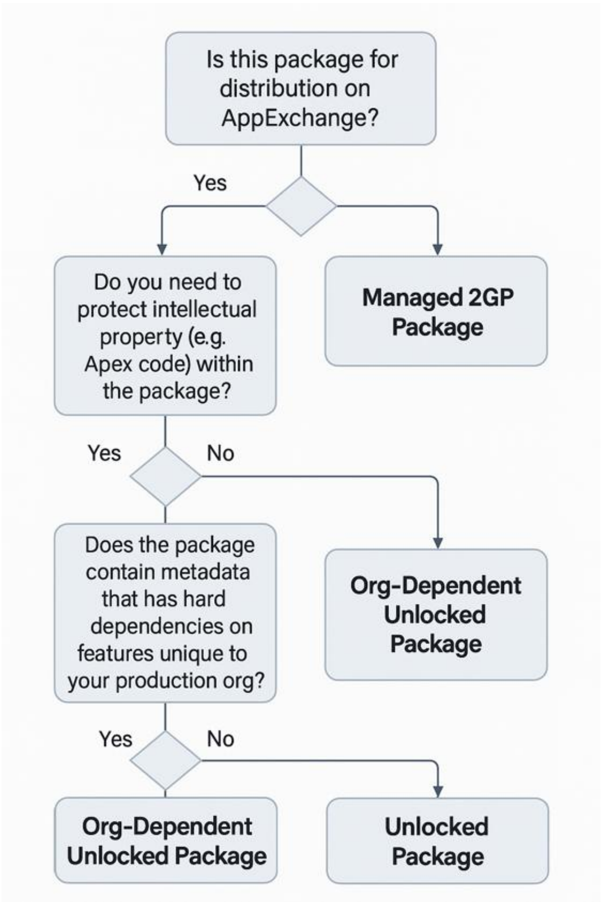
Is this package for distribution on AppExchange?
Yes
Do you need to protect intellectual property (e.g. Apex code) within the package?
Managed 2GP Package
Yes
No
Does the package contain metadata that has hard dependencies on features unique to your production org?
Org-Dependent Unlocked Package
Yes
No
Org-Dependent Unlocked Package
Unlocked Package

Fig. 2. Decision Tree for Selecting the Right Salesforce Package Type

C. Measuring Success: Key Performance Indicators for Delivery Optimization

Organizations need to apply industry-standard metrics to change subjective perceptions of "moving faster" into objective, data-based strategies for improvement.

1) *Introducing DORA Metrics:* The DevOps Research and Assessment (DORA) program has identified four key performance metrics that serve as universal standards for evaluating the effectiveness of software delivery teams. These metrics provide a comprehensive overview of production output and operational stability.

The four key DORA metrics are:

- **Deployment Frequency:** The measurement quantifies the number of times organizations deploy code to production environments. Throughput directly correlates with this metric. Elite-performing teams deploy code on demand at rates reaching multiple deployments per day.
- **Lead Time for Changes:** The time taken from a committed change to its implementation in production is a key metric. This indicator plays a vital role in determining the efficiency of the development and release process. Leading teams are capable of achieving a lead time of less than one hour.

Change Failure Rate (CFR): This metric reflects the percentage of production deployments that either cause system failures or require urgent hotfixes. This text is a key measure of quality and stability in team performance. Successful teams keep their failure rates between 0% and 15%

- **Mean Time to Recovery (MTTR):** This metric looks at the time it takes to restore service following a production failure, reflecting the team's ability to recover from disruptions. High achievers can recover from these failures in less than one hour. [4]

Context from the State of Salesforce DevOps Reports

The Salesforce ecosystem benefits from metric analysis for obtaining meaningful insights. The 2021 State of Salesforce DevOps report demonstrated that the ecosystem faced quality and stability declines because change failure rates increased while recovery times lengthened. The pandemic-related disruptions caused organizations to adopt DevOps tools for speed, but this needs to be balanced with a dedicated quality focus. Data reveals that automatic pre-deployment testing leads to better stability because teams using this practice achieve significant reductions in change failure rates. [4] The essential part of any optimization strategy involves integrating automated testing into the CI/CD pipeline, according to this finding.

VI. CONCLUSION

A. Summary of Findings

The research investigates Salesforce development and deployment practices through systematic analysis to establish successful methods for improving delivery cycles. The study demonstrates that organizations must undergo technological and cultural changes to achieve higher velocity and stability.

The study's key findings are summarized as follows:

- The Software Development Lifecycle, which utilizes 1GP packaging and manual deployment tools, including Change Sets and the Ant Migration Tool, operates with fundamental limitations. The model features manual, error-prone operations, which create high deployment failure risks while lacking traceability and producing systematic bottlenecks that naturally block quick and reliable delivery.
- The Salesforce DX tool suite, together with 2GP packaging, represents essential architectural and philosophical advancement for modern development. The contemporary approach establishes version control systems as the central source of truth, replacing Salesforce orgs, which enables DevOps principles and automated collaboration, and prediction needed for large-scale agile development.
- Successful optimization requires more than a single tool or process because it needs a complete strategic approach. The successful strategy requires three essential components: appropriate technology (2GP for distribution and Unlocked Packages for internal modularity and Salesforce DX for automation), suitable processes (a balanced release strategy for patches and major versions and CI/CD pipelines with

integrated testing), and appropriate culture (a "packaging mindset" with modularity focus and teamwide quality and collaboration commitment).

B. Final Recommendation

Organizations seeking to accelerate their Salesforce delivery cycle should implement a source-driven development culture as the primary recommendation from this paper. All subsequent optimizations stem from this fundamental change.

The first step should be to establish a version control system like Git as the only source of truth for all metadata changes for both declarative and programmatic development. Organizations should first create a base system before implementing tools and practices that support the new paradigm. The Salesforce CLI serves as the primary development tool and automation interface. To structure the application architecture, 2GP or Unlocked Packages should be utilized. Additionally, a complete CI/CD pipeline should be implemented to automate build, testing, and deployment processes.

The complete transformation will enable organizations to move beyond their current development model, which produces obstacles, risks, and delays. By transforming its Salesforce development practice, the organization can shift from an organizational limitation to a business agility enabler that delivers innovative solutions to users and customers at the market-required speed and quality.

REFERENCES:

[1] B. McCarthy, “Complete Guide to Salesforce DevOps,” *Salesforce Ben*, Feb. 01, 2021. Available: https://www.salesforceben.com/salesforce-devops/

[2] S. Architects, “Platform Multitenant architecture,” Aug. 01, 2022. Available: https://architect.salesforce.com/fundamentals/platform-multitenant-architecture

[3] A. Khandelwal, “7 phases of Salesforce Development lifecycle,” *ITChronicles*, Sep. 01, 2022. Available: https://itchronicles.com/software-development/7-phases-of-salesforce-development-lifecycle/

[4] “2022 State of Salesforce DevOps Report | COPADO.” Available: https://www.copado.com/resources/reports/2022-state-of-salesforce-devops-report

[5] salesforcedocs, “ISVForce Guide,” book, Aug. 2022. Available: https://resources.docs.salesforce.com/238/latest/en-us/sfdc/pdf/salesforce_packaging_guide.pdf

[6] “Help and training community,” *Salesforce*, Oct. 13, 2022. Available: https://help.salesforce.com/s/articleView?id=000385206&language=en_US&type=1

[7] Mirketa, “A complete guide to Second-Generation packages in Salesforce,” *mirketa*, Apr. 04, 2022. Available: https://mirketa.com/second-generation-packages-in-salesforce/

[8] “Release Strategies for Salesforce Managed Packages: Our Experience,” *Ascendix Technologies*, Sep. 23, 2022. Available: https://ascendix.com/blog/salesforce-managed-package-release-strategies/

[9] P. Kumar, “All you need to know about Salesforce Development Lifecycle,” *Algoworks*, May 14, 2015. Available: https://www.algoworks.com/blog/salesforce-development-lifecycle/

[10] S. Nikam, “Tips for Salesforce Release Managers to Ensure Successful Deployments,” *Focus On Force*. Available: https://focusonforce.com/crm/tips-for-salesforce-release-managers-to-ensure-successful-deployments/

[11] Apex Hours, “Development and deployment process,” *Apex Hours*, Aug. 18, 2021. Available: https://www.apexhours.com/development-and-deployment-process/

[12] A. Chaudhary, “Deployment using change sets in Salesforce,” *Apex Hours*, Oct. 31, 2022. Available: https://www.apexhours.com/deployment-using-change-sets-in-salesforce/

[13] Pavan, “Salesforce Deployment – A high level overview,” *Sfdc Techie - Pavan’s Blog*, Jul. 01, 2018. Available: https://sfdctechie.wordpress.com/2018/07/01/salesforce-deployment-a-high-level-overview/

[14] A. Sripathmathasan, “Salesforce ANT Migration Tool,” *Medium*, Aug. 28, 2021. Available: https://medium.com/@anand_19/salesforce-ant-migration-tool-8d07799c7a2f

[15] Salesforce, “Salesforce Developers.” Available: https://developer.salesforce.com/docs/atlas.en-us.238.0.daas.meta/daas/meta_development.htm
[16] Jitendra, “Complete Salesforce Deployment Guide using Ant Migration Tool,” *Jitendra Zaa*. Available: https://www.jitendrazaa.com/blog/salesforce/salesforce-migration-tool-ant/
[17] Jitendra, “Continuous integration in Salesforce Using Jenkins and Git | Video Tutorial,” *Jitendra Zaa*. Available: https://www.jitendrazaa.com/blog/salesforce/continuous-integration-in-salesforce-using-jenkins-and-git-video-tutorial/
[18] Team Absyz, “Salesforce deployment using ANT migration tool,” *Absyz*. Available: https://www.absyz.com/salesforce-deployment-using-ant-migration-tool/
[19] R. Seroter, “Salesforce.com introduces extensive changes to developer experience,” *InfoQ*, Oct. 06, 2016. Available: https://www.infoq.com/news/2016/10/salesforce-dx/
[20] M. Dickens, “Salesforce DX - the new Salesforce developer experience,” *Gearset*, Oct. 26, 2016. Available: https://gearset.com/blog/salesforce-dx/
[21] Salesforce, “Introducing the Salesforce DX Open Beta,” *Salesforce Developers Blog*, Jun. 28, 2017. Available: https://developer.salesforce.com/blogs/developer-relations/2017/06/introducing-salesforce-dx-open-beta.