

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA CÔNG NGHỆ PHẦN MỀM



Seminar các vấn đề hiện đại của CNPM - SE400

XÂY DỰNG QUY TRÌNH CI/CD PIPELINE DEVSECOPS
VỚI DOCKER, GITLAB, KUBERNETES

GV HƯỚNG DẪN: ThS. Đinh Nguyễn Anh Dũng
SINH VIÊN THỰC HIỆN: Lê Văn Duy – 21521997
LỚP: SE400.P11.PMCL

TP. HỒ CHÍ MINH, 2024

TÓM TẮT ĐỒ ÁN

Đề tài của tôi xoay quanh việc xây dựng một quy trình DevSecOps CI/CD toàn diện cho dự án ứng dụng web kết hợp giữa React và Go, với việc áp dụng các công nghệ tiên tiến như Docker, GitLab CI và Kubernetes. Mục tiêu chính của đồ án là tạo ra một hệ thống tự động hóa quy trình phát triển phần mềm, từ việc viết mã đến triển khai và duy trì, đồng thời tích hợp các yếu tố bảo mật ngay từ đầu.

Trong bài báo cáo, tôi sẽ trình bày cơ sở lý thuyết về DevOps và CI/CD, cùng với việc làm rõ tầm quan trọng của tư duy DevSecOps, điều này giúp đảm bảo rằng bảo mật không chỉ là một bước cuối cùng mà là một phần không thể thiếu trong toàn bộ quy trình phát triển. Tôi sẽ đi sâu vào các công cụ bảo mật như SAST (Static Application Security Testing), SCA (Software Composition Analysis), Image Scan và DAST (Dynamic Application Security Testing). Những công cụ này sẽ giúp phát hiện và xử lý các lỗ hổng bảo mật ngay trong quá trình phát triển, từ đó nâng cao độ an toàn cho ứng dụng.

Ngoài ra, tôi sẽ trình bày cách xây dựng một pipeline CI/CD hiệu quả, đảm bảo tích hợp và triển khai tự động. Phần này sẽ bao gồm cả việc kiểm tra hiệu suất để đảm bảo rằng ứng dụng không chỉ hoạt động an toàn mà còn hoạt động hiệu quả.

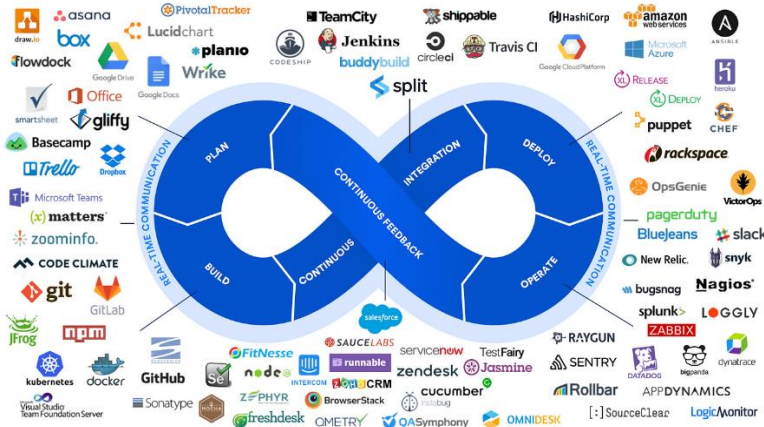
Kubernetes sẽ được sử dụng chủ yếu để thiết lập môi trường production, tuy nhiên, tôi sẽ chỉ tập trung vào những kiến thức cơ bản cần thiết để triển khai ứng dụng trên nền tảng này, mà không đi sâu quá nhiều vào các khái niệm phức tạp của Kubernetes. Qua đó, tôi hy vọng có thể cung cấp một cái nhìn tổng quan và thực tiễn về quy trình DevSecOps, giúp người đọc hiểu rõ hơn về những lợi ích của việc áp dụng phương pháp này trong phát triển phần mềm hiện đại.

MỤC LỤC

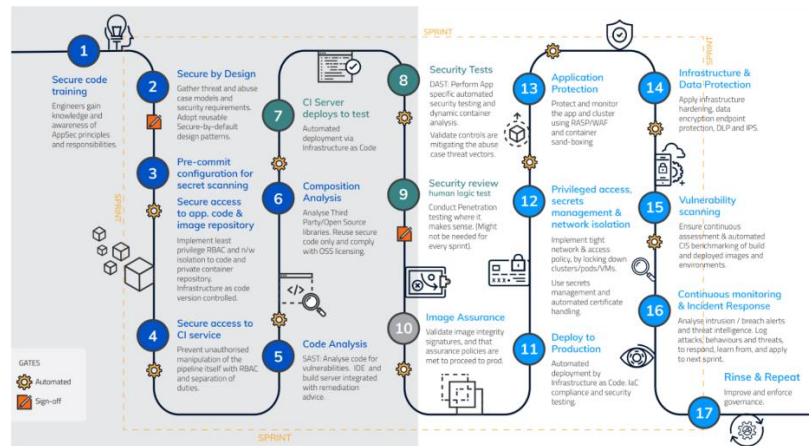
TÓM TẮT ĐỒ ÁN	1
MỤC LỤC	2
I. Giới thiệu	3
1. Tầm quan trọng của DevSecOps và CI/CD trong phát triển phần mềm hiện đại	3
2. Mục đích và phạm vi nghiên cứu	5
II. Khái niệm về DevOps, DevSecOps và CI/CD	6
1. DevOps và DevSecOps	6
2. Continuous Integration và Continuous Delivery (CI/CD)	10
3. Quy trình triển khai CI/CD và DevSecOps	12
4. Các loại test trong DevSecOps	16
III. Các công cụ và công nghệ sử dụng trong quy trình	19
1. Docker	19
2. GitLab và GitLab CI.....	23
3. Xây dựng Pipeline với GitLab CI và Docker	29
4. Kubernetes	31
5. GitOps và ArgoCD	39
IV. Xây dựng quy trình CI/CD Pipeline toàn diện DevSecOps	45
1. Thiết kế quy trình CI/CD DevSecOps.....	45
2. Thiết Lập Môi Trường Phát Triển và CI/CD.....	46
3. Các Bước Xây Dựng Quy trình CI/CD Thực tế	47
V. Kết luận và hướng phát triển	48
1. Tóm tắt quy trình CI/CD và DevSecOps.....	48
2. Những thách thức và giải pháp khi triển khai DevSecOps trong dự án.....	48
3. Hướng phát triển trong tương lai.....	49
TÀI LIỆU THAM KHẢO	51

I. Giới thiệu

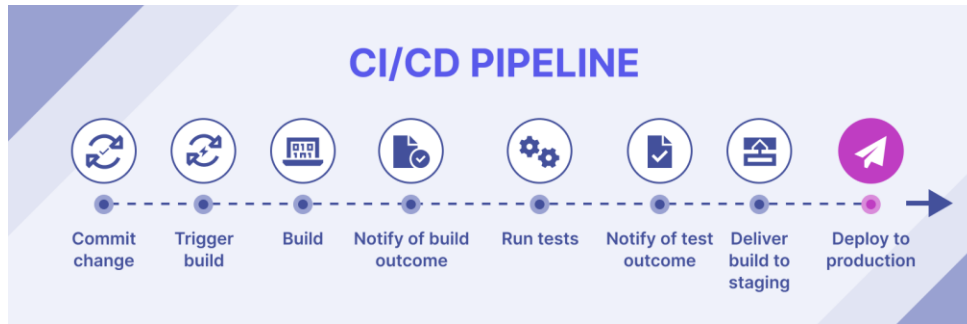
1. Tầm quan trọng của DevSecOps và CI/CD trong phát triển phần mềm hiện đại



DevOps là một phương pháp phát triển phần mềm giúp tối ưu hóa quy trình phát triển và triển khai ứng dụng thông qua việc kết hợp các hoạt động của đội ngũ phát triển (Dev) và đội ngũ vận hành (Ops). Mục tiêu của DevOps là giảm thiểu thời gian triển khai phần mềm, cải thiện chất lượng và khả năng phản hồi nhanh chóng với các thay đổi yêu cầu từ người dùng. Tuy nhiên, một vấn đề quan trọng trong quá trình phát triển phần mềm là bảo mật.



DevSecOps (Development, Security, and Operations) ra đời để giải quyết vấn đề này, tức là tích hợp các biện pháp bảo mật ngay từ đầu trong quá trình phát triển phần mềm, thay vì chờ đến khi phần mềm đã sẵn sàng để triển khai mới nghĩ đến bảo mật. DevSecOps khuyến khích việc áp dụng bảo mật tự động, kiểm tra mã nguồn và các hình thức bảo mật khác vào từng giai đoạn trong quy trình DevOps.



CI/CD (Continuous Integration / Continuous Deployment) là một phần quan trọng trong mô hình DevOps. CI giúp tự động hóa quá trình tích hợp mã nguồn từ các lập trình viên vào hệ thống kiểm thử và sản xuất. CD giúp tự động hóa việc triển khai mã nguồn mới lên môi trường production. Việc tích hợp bảo mật vào các pipeline này giúp giảm thiểu các lỗ hổng bảo mật và đảm bảo an toàn cho hệ thống từ khi phát triển cho đến khi sản phẩm được triển khai ra môi trường thực tế.

Lợi ích của việc tích hợp bảo mật vào quy trình DevOps (DevSecOps):

- Tăng cường bảo mật sớm: Việc tích hợp bảo mật từ giai đoạn đầu giúp phát hiện và khắc phục lỗi bảo mật ngay trong quá trình phát triển, thay vì chỉ kiểm tra bảo mật khi ứng dụng đã hoàn thành.
- Giảm thiểu rủi ro: Tự động kiểm tra bảo mật liên tục giúp phát hiện và giảm thiểu các mối đe dọa từ bên ngoài.
- Tiết kiệm thời gian và chi phí: Việc phát hiện và sửa lỗi sớm trong quá trình phát triển sẽ giúp tiết kiệm chi phí khắc phục sự cố sau khi triển khai.
- Cải thiện chất lượng phần mềm: Quy trình CI/CD DevSecOps giúp ứng dụng phần mềm được kiểm tra, cập nhật và triển khai nhanh chóng, giảm thiểu sự cố và đảm bảo chất lượng phần mềm.

Các yếu tố thúc đẩy sự phát triển của DevSecOps và CI/CD:

- Tăng trưởng của các mối đe dọa bảo mật: Trong thời đại số, các cuộc tấn công mạng ngày càng tinh vi và phức tạp. Việc tích hợp bảo mật ngay từ đầu giúp đối phó hiệu quả hơn với các mối đe dọa này.
- Yêu cầu về tính linh hoạt: Các doanh nghiệp cần phát triển và triển khai phần mềm nhanh chóng để đáp ứng nhu cầu thị trường. CI/CD là giải pháp giúp tiết kiệm thời gian và chi phí trong quá trình này.
- Chuyển đổi số trong doanh nghiệp: Việc chuyển sang môi trường đám mây và các công nghệ container như Docker, Kubernetes tạo ra yêu cầu về một quy trình triển khai tự động và bảo mật.

2. Mục đích và phạm vi nghiên cứu

Bài báo cáo này nhằm xây dựng quy trình DevSecOps CI/CD pipeline cho ứng dụng React + Go, giúp triển khai và kiểm thử tự động ứng dụng trong môi trường an toàn và hiệu quả. Quy trình này sẽ bao gồm việc sử dụng Docker để đóng gói ứng dụng, GitLab CI để tự động hóa quy trình kiểm tra và triển khai, và Kubernetes để triển khai ứng dụng lên môi trường đám mây.

Phạm vi nghiên cứu của bài báo cáo chủ yếu tập trung vào việc triển khai quy trình DevSecOps và CI/CD cho một ứng dụng web xây dựng bằng React (frontend) và Go (backend). Các công nghệ và công cụ sử dụng bao gồm:

- Docker: Được sử dụng để tạo ra các container cho ứng dụng, giúp việc triển khai trở nên dễ dàng và nhanh chóng. Docker cũng giúp tạo ra môi trường ổn định và giống nhau cho cả phát triển và sản xuất.
- GitLab CI: Là công cụ giúp tự động hóa các bước trong quy trình phát triển, kiểm thử và triển khai phần mềm. GitLab CI cung cấp các pipeline để tự động hóa các nhiệm vụ như kiểm tra mã nguồn, chạy unit test, kiểm tra bảo mật, và triển khai ứng dụng.
- Kubernetes: Là công cụ giúp quản lý và triển khai các ứng dụng container. Kubernetes hỗ trợ việc mở rộng quy mô ứng dụng một cách linh hoạt và an toàn, cũng như giúp tự động hóa việc triển khai ứng dụng lên môi trường production.

Các vấn đề sẽ được đề cập:

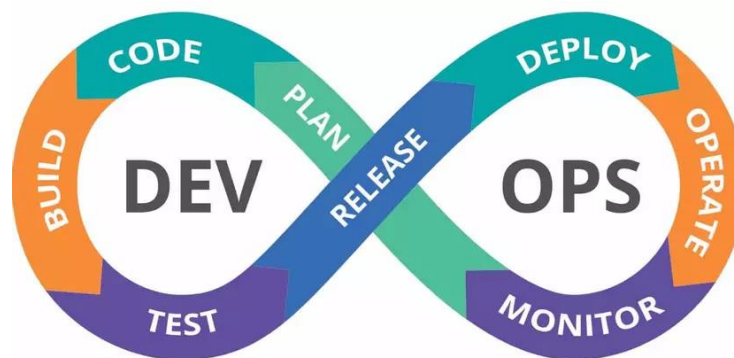
- Quy trình triển khai DevOps: Hướng dẫn chi tiết các bước xây dựng một quy trình DevOps từ đầu đến cuối, bao gồm tích hợp liên tục (CI) và triển khai liên tục (CD).
- Các loại test trong DevSecOps: Giới thiệu các loại kiểm thử quan trọng trong DevSecOps như static code analysis, vulnerability scanning, security testing, và penetration testing.
- Cách xây dựng và tối ưu hóa pipeline: Tập trung vào cách thiết lập một pipeline CI/CD hiệu quả, tối ưu hóa để đảm bảo chất lượng phần mềm, giảm thiểu lỗi và đảm bảo bảo mật cho ứng dụng trong mọi giai đoạn.

II. Khái niệm về DevOps, DevSecOps và CI/CD

1. DevOps và DevSecOps

1.1. DevOps

DevOps là một **phương pháp phát triển phần mềm** kết hợp giữa Development (phát triển) và Operations (vận hành), nhằm rút ngắn chu kỳ phát triển và cung cấp phần mềm nhanh chóng, ổn định. Khái niệm DevOps lần đầu xuất hiện vào cuối những năm 2000 và trở nên phổ biến từ năm 2009, khi các tổ chức nhận thấy cần tăng cường sự hợp tác giữa các nhóm phát triển và nhóm vận hành. Trước đây, hai nhóm này thường làm việc tách biệt, gây ra nhiều khó khăn trong việc triển khai phần mềm.



DevOps **không chỉ là một bộ công cụ** mà là một **văn hóa làm việc**, kết hợp các kỹ sư phát triển phần mềm (Dev) với các kỹ sư hệ thống, bảo mật, mạng, hạ tầng (Ops). Mục tiêu là **tăng tốc độ phát triển và chuyển giao sản phẩm, cải tiến sản phẩm nhanh hơn**, từ đó giúp tổ chức phục vụ khách hàng hiệu quả hơn và cạnh tranh tốt hơn trên thị trường.

Bằng cách phá vỡ rào cản giữa các bộ phận, DevOps tạo ra một môi trường hợp tác chặt chẽ, trong đó tất cả các thành viên cùng làm việc vì mục tiêu chung, nâng cao hiệu suất và độ ổn định của phần mềm. **Đây không phải là một quy trình cứng nhắc hay tiêu chuẩn phải tuân theo, mà là một phương pháp làm việc linh hoạt**, yêu cầu sự hỗ trợ của các công cụ để đạt được sự phối hợp và tự động hóa hiệu quả trong toàn bộ vòng đời phát triển phần mềm.

Các nguyên tắc chính của DevOps:

- Tự động hóa: Các quy trình, từ viết mã, kiểm thử đến triển khai, đều được tự động hóa giúp tăng tốc độ và giảm thiểu lỗi do con người gây ra.
- Hợp tác: Khuyến khích sự hợp tác chặt chẽ giữa các nhóm phát triển và vận hành để cùng làm việc với mục tiêu chung.
- Liên tục: DevOps đặt nặng yếu tố liên tục trong tất cả các khía cạnh như tích hợp liên tục (CI), triển khai liên tục (CD), giúp sản phẩm có thể cập nhật thường xuyên.

- Phản hồi nhanh: Thu thập phản hồi từ người dùng và các công cụ giám sát giúp phát hiện và xử lý lỗi nhanh chóng, tăng trải nghiệm người dùng.

1.2. Các khía cạnh liên quan và tư duy về DevOps và CI/CD

DevOps là một phương pháp tiếp cận trong việc phát triển phần mềm nhằm tối ưu hóa toàn bộ quá trình từ phát triển đến vận hành. Một trong những yếu tố quan trọng nhất của DevOps là **tư duy hợp tác**. Trước khi có DevOps, các nhóm phát triển, kiểm thử và vận hành thường làm việc độc lập, đôi khi gây ra hiểu lầm và mất thời gian khi chuyển giao công việc giữa các nhóm. Với DevOps, các nhóm này không còn tách biệt nữa mà hợp tác chặt chẽ trong suốt vòng đời của phần mềm.

- Nhóm phát triển (Development): Chịu trách nhiệm xây dựng và viết mã nguồn cho phần mềm.
- Nhóm kiểm thử (Testing): Đảm bảo chất lượng phần mềm thông qua các bài kiểm thử, phát hiện lỗi.
- Nhóm vận hành (Operations): Đảm bảo phần mềm chạy ổn định trên môi trường sản xuất.

Môi quan hệ hợp tác này giúp tăng tốc độ phát triển và cải thiện chất lượng phần mềm.

Một triết lý quan trọng trong DevOps là **"fail fast, fail early"** (thất bại nhanh, thất bại sớm). Ý nghĩa của câu này là khuyến khích việc phát hiện lỗi sớm trong quá trình phát triển, thay vì để lỗi kéo dài đến giai đoạn cuối hoặc sau khi phần mềm đã được triển khai. Khi phát hiện lỗi sớm, các nhóm phát triển có thể sửa chữa nhanh chóng, giảm thiểu tác động tiêu cực đến toàn bộ dự án.

Triết lý này được thực hiện thông qua việc tự động hóa kiểm thử, liên tục kiểm tra và phản hồi từ quá trình tích hợp liên tục (CI) và triển khai liên tục (CD). Điều này giúp phát hiện các lỗi tiềm ẩn ngay từ khi mã được viết, trước khi chúng trở thành vấn đề lớn.

DevOps không chỉ là một phương pháp mà còn là một **văn hóa cải tiến liên tục (Continuous Improvement)**. Điều này có nghĩa là các nhóm không bao giờ ngừng cải tiến quy trình, công cụ, và kỹ thuật của mình. Mỗi giai đoạn trong vòng đời phát triển phần mềm đều có thể được cải tiến để làm việc hiệu quả hơn, giảm thiểu lỗi và tăng tốc độ triển khai.

Cải tiến liên tục không chỉ giúp nâng cao hiệu suất mà còn khuyến khích các nhóm học hỏi từ những sai lầm và cải tiến theo từng ngày. Ví dụ, qua mỗi lần triển khai, đội DevOps sẽ đánh giá lại quy trình và tìm cách để làm cho công việc nhanh chóng và hiệu quả hơn trong lần sau.

Các khía cạnh kỹ thuật của DevOps:

1. Quản lý mã nguồn và phân phối phần mềm

Trong DevOps, việc quản lý mã nguồn rất quan trọng vì nó liên quan trực tiếp đến việc phát triển, kiểm thử, và triển khai phần mềm. Các công cụ quản lý mã nguồn phổ biến như **Git** cho phép các nhóm làm việc trên cùng một mã nguồn mà không gặp phải xung đột. Mỗi thay đổi trong mã nguồn sẽ được ghi lại và có thể theo dõi, giúp giảm thiểu rủi ro khi triển khai phần mềm.

Các hệ thống CI/CD như **GitLab** hoặc **Jenkins** sẽ tự động hóa quá trình xây dựng (build) và kiểm thử phần mềm mỗi khi có thay đổi mã nguồn. Điều này giúp phát hiện lỗi và tăng tốc độ phát triển phần mềm.

2. Tự động hóa kiểm thử và kiểm tra chất lượng mã nguồn

Kiểm thử tự động là một phần quan trọng trong DevOps. Thay vì chờ đợi đến giai đoạn cuối để kiểm tra phần mềm, các kiểm thử tự động được chạy liên tục trong quá trình phát triển. Các công cụ như **Selenium**, **JUnit**, hoặc **TestNG** sẽ giúp kiểm tra tính năng và chất lượng mã nguồn mỗi khi có sự thay đổi. Điều này không chỉ giúp phát hiện lỗi sớm mà còn giảm thiểu công sức của đội ngũ kiểm thử.

Kiểm tra chất lượng mã nguồn cũng rất quan trọng, đặc biệt khi dự án lớn và mã nguồn phức tạp. Các công cụ như **SonarQube** giúp phân tích chất lượng mã nguồn, tìm ra các vấn đề như lỗi cú pháp, code smell, hoặc các lỗ hổng bảo mật.

3. Giám sát và theo dõi hiệu suất ứng dụng

Giám sát ứng dụng là một phần không thể thiếu trong DevOps. Sau khi phần mềm được triển khai, việc theo dõi và giám sát liên tục sẽ giúp các nhóm phát hiện sớm các vấn đề hiệu suất hoặc sự cố hệ thống. Các công cụ như **Prometheus**, **Grafana**, hay **Datadog** sẽ giúp thu thập và hiển thị dữ liệu về hiệu suất ứng dụng, giúp đội ngũ vận hành đưa ra quyết định kịp thời.

Ví dụ, nếu hệ thống nhận thấy có sự gia tăng sử dụng CPU hoặc bộ nhớ, các cảnh báo có thể được gửi đến đội ngũ vận hành để họ có thể xử lý sự cố trước khi nó ảnh hưởng đến người dùng cuối.

4. Quản lý cấu hình và môi trường triển khai

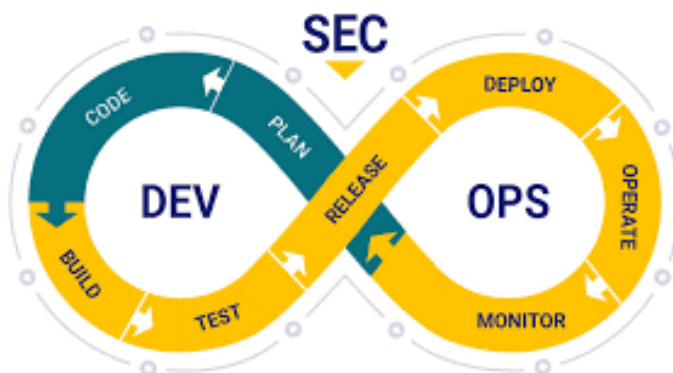
Quản lý cấu hình và môi trường triển khai là một yếu tố quan trọng khác trong DevOps. Việc cấu hình môi trường phát triển, kiểm thử và sản xuất sao cho nhất quán và tự động hóa sẽ giúp giảm thiểu rủi ro lỗi khi triển khai. Các công cụ như **Docker** và **Kubernetes** hỗ trợ việc quản lý cấu hình và môi trường triển khai.

- **Docker:** Giúp đóng gói ứng dụng và các phụ thuộc vào một container độc lập, đảm bảo ứng dụng chạy nhất quán trên mọi môi trường.
- **Kubernetes:** Quản lý và tự động hoá việc triển khai các container, giúp ứng dụng dễ dàng mở rộng và duy trì trên nhiều máy chủ.

Thông qua việc sử dụng các công cụ này, quá trình triển khai và quản lý ứng dụng trở nên đơn giản và hiệu quả hơn, giúp giảm thiểu lỗi do sự khác biệt giữa các môi trường.

1.3. DevSecOps

DevSecOps là **bước phát triển tiếp theo của DevOps khi tích hợp bảo mật (Security)** vào các giai đoạn phát triển phần mềm. Trước đây, bảo mật thường chỉ được xem xét ở cuối quá trình phát triển, dẫn đến việc xử lý lỗi bảo mật mất nhiều thời gian và chi phí. DevSecOps giúp các nhóm phát triển phần mềm xây dựng ứng dụng vừa an toàn vừa hiệu quả ngay từ đầu, không phải đợi đến giai đoạn cuối mới kiểm tra bảo mật.



Chúng ta cần đặt ra câu hỏi là **Tại sao bảo mật lại quan trọng trong DevOps?** Khi phần mềm được phát triển và triển khai nhanh chóng, các vấn đề bảo mật có thể bị bỏ qua hoặc phát hiện muộn, gây rủi ro lớn. Trong bối cảnh các cuộc tấn công mạng ngày càng phức tạp, bảo mật đóng vai trò quan trọng để bảo vệ dữ liệu và uy tín của doanh nghiệp. DevSecOps đảm bảo bảo mật không chỉ là trách nhiệm của nhóm bảo mật mà là của cả đội ngũ phát triển, giúp phát hiện sớm và xử lý kịp thời các lỗ hổng tiềm ẩn.

Trong DevSecOps, bảo mật được tích hợp vào các giai đoạn chính của quy trình DevOps:

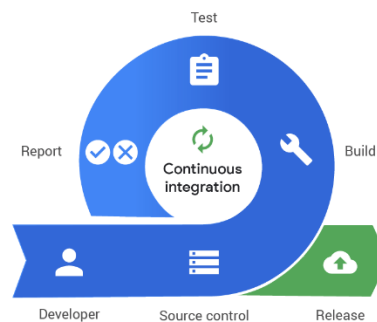
1. **Lập kế hoạch và thiết kế:** Xác định các rủi ro bảo mật từ giai đoạn đầu.
2. **Viết mã:** Thực hiện các biện pháp bảo mật trong việc viết mã, bao gồm cả kiểm tra mã tự động để phát hiện lỗ hổng.
3. **Kiểm thử:** Sử dụng các công cụ kiểm thử bảo mật tự động như quét mã, kiểm tra lỗ hổng trong thư viện bên thứ ba.
4. **Triển khai:** Quá trình triển khai được bảo mật bằng cách sử dụng các công cụ giám sát, phát hiện xâm nhập để phát hiện bất kỳ dấu hiệu bất thường nào.

5. **Vận hành và bảo trì:** Giám sát liên tục, cập nhật các bản vá bảo mật, duy trì tính bảo mật của phần mềm trong suốt vòng đời.

2. Continuous Integration và Continuous Delivery (CI/CD)

2.1. CI/CD là gì?

Continuous Integration (CI), hay tích hợp liên tục, là một phương pháp trong phát triển phần mềm cho phép các lập trình viên thường xuyên đưa mã mới vào nhánh chính của dự án. Khi có mã mới được tích hợp, hệ thống CI sẽ tự động kiểm tra để đảm bảo không có lỗi phát sinh và mã vẫn hoạt động đúng. CI giúp phát hiện sớm lỗi khi mã từ nhiều thành viên được kết hợp lại, tránh tình trạng "tích hợp vào phút cuối" thường dẫn đến nhiều lỗi phát sinh cùng lúc.



Lợi ích của việc sử dụng CI:

- Giảm thiểu rủi ro nhờ việc phát hiện lỗi và fix sớm, tăng chất lượng phần mềm nhờ việc tự động test và inspect (đây cũng là một trong những lợi ích của CI, code được inspect tự động dựa theo config đã cài đặt, đảm bảo coding style, chẳng hạn một function chỉ được dài không quá 10 dòng code ...)
- Giảm thiểu những quy trình thủ công lặp đi lặp lại (build css, js, migrate, test...), thay vì đó là build tự động, chạy test tự động
- Sinh ra phần mềm có thể deploy ở bất kì thời gian, địa điểm

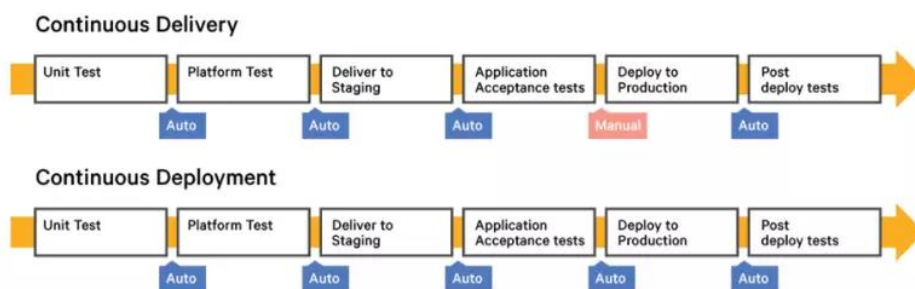
Continuous Delivery (CD) là một bước tiến tiếp theo sau CI, là phương pháp tự động hóa quá trình triển khai phần mềm. Sau khi mã nguồn được kiểm tra và xây dựng qua CI, CD đảm bảo rằng phần mềm luôn sẵn sàng để triển khai lên môi trường production bất kỳ lúc nào. Mặc dù CD tự động hóa việc triển khai phần mềm, tuy nhiên vẫn có thể cần sự can thiệp của con người để quyết định khi nào chính thức đưa phần mềm vào môi trường production. Đó là lý do vì sao có sự phân biệt giữa Continuous Delivery và Continuous Deployment.

Có một khái niệm nữa là Continuous Deployment, và hai khái niệm này thường hay bị nhầm lẫn với nhau. Nếu Continuous Delivery là triển khai code lên môi trường staging, và

deploy thủ công lên môi trường production, thì Continuous Deployment (cũng viết tắt là CD) lại là kỹ thuật để triển khai code lên môi trường production một cách tự động, và cũng nên là mục tiêu của hầu hết công ty.

Về cơ bản thì môi trường staging là môi trường giống với production, nên đã làm Continuous Delivery được thì cũng làm Continuous Deployment được. Tuy nhiên, thực tế lại không dễ dàng như vậy. Lý do thứ nhất là chúng ta có thể deploy tự động lên staging, nhưng liệu chúng ta có dám deploy tự động với production, cho dù là mọi cấu hình đều giống nhau thì thực tế staging và production server vẫn là hai server riêng biệt, và vì thế không thể đảm bảo mọi thứ chạy đúng trên staging sẽ chạy đúng trên production, thế nên deploy lên production thường phải làm thủ công để chắc chắn là các bước build, test được thực hiện chính xác. Lý do thứ hai đơn giản hơn, đó là rất khó để test tự động hoàn toàn, và bởi vậy khó mà tự động deploy được.

Dù Continuous Deployment có thể không phù hợp với mọi công ty, nhưng Continuous Delivery thì tuyệt đối là yêu cầu cho việc thực hiện triết lý DevOps. Chỉ khi code được chuyển giao liên tục, chúng ta mới có thể tự tin rằng những thay đổi từ code sẽ phục vụ cho khách hàng sau chỉ vài phút với một nút ấn.



CI và CD đều đóng vai trò quan trọng trong việc tăng cường hiệu quả của quy trình phát triển phần mềm. CI giúp đảm bảo rằng mã nguồn được tích hợp liên tục và luôn ở trạng thái hoạt động tốt (không có lỗi), trong khi CD đảm bảo rằng phần mềm luôn sẵn sàng để triển khai lên môi trường thực tế.

Nói cách khác, **CI là bước tiền đề để thực hiện CD**. Khi mọi thay đổi trong mã nguồn đều được kiểm tra tự động và xác nhận không có lỗi, phần mềm có thể nhanh chóng được triển khai lên môi trường thử nghiệm hoặc sản xuất mà không cần phải lo lắng về vấn đề lỗi phát sinh.

2.2. Các lợi ích của CI/CD

CI/CD giúp **tự động hóa** các bước từ kiểm thử đến triển khai. Nhờ có các công cụ CI/CD, như GitLab CI, Jenkins, các bước kiểm tra và tích hợp sẽ được thực hiện tự động

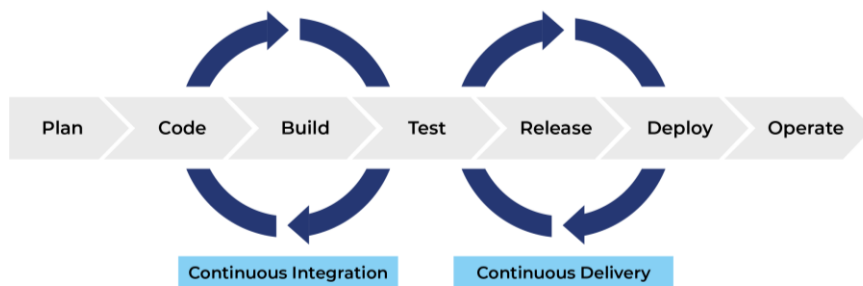
ngay khi có mã mới. Điều này giảm thiểu lỗi do quá trình thủ công gây ra và đảm bảo mã luôn được kiểm thử một cách nhất quán.

Khi tích hợp mã thường xuyên, **các lỗi được phát hiện sớm hơn** và dễ dàng sửa chữa hơn. CI/CD giúp giảm thời gian chờ đợi khi kiểm tra mã theo cách thủ công, tăng tốc độ phát triển và cải thiện hiệu suất làm việc của đội ngũ lập trình viên. Thay vì chờ đợi lâu, các thay đổi nhỏ được kiểm thử và đưa vào sản phẩm liên tục, giúp tối ưu hóa thời gian phát triển.

Trong bối cảnh bảo mật ngày càng quan trọng, CI/CD có thể tích hợp các công cụ và quy trình DevSecOps để thực hiện kiểm tra bảo mật tự động. Các công cụ như SonarQube, OWASP Dependency-Check có thể được tích hợp vào pipeline để kiểm tra các lỗ hổng bảo mật ngay từ giai đoạn kiểm thử. Điều này giúp phần mềm được bảo mật liên tục, giảm thiểu rủi ro bảo mật khi triển khai.

3. Quy trình triển khai CI/CD và DevSecOps

3.1. Các bước triển khai CI/CD: Từ code commit đến production



Quy trình CI/CD giúp tự động hóa các công đoạn kiểm tra, xây dựng và triển khai phần mềm từ giai đoạn phát triển (code commit) đến môi trường sản xuất. Dưới đây là các bước chính trong quy trình CI/CD:

1. Code Commit:

- Lập trình viên viết mã nguồn và **commit** vào hệ thống quản lý mã nguồn như **GitLab**. Mỗi lần có một **commit** mới vào nhánh chính (main branch), quy trình CI sẽ tự động được kích hoạt để bắt đầu kiểm tra và xây dựng lại ứng dụng.

2. Continuous Integration (CI):

- **Xây dựng (Build):** Sau khi commit mã nguồn, hệ thống CI sẽ tự động tải mã nguồn và xây dựng lại ứng dụng để kiểm tra xem mã mới có gây lỗi trong quá trình biên dịch hay không. Nếu xây dựng thành công, hệ thống sẽ tiếp tục kiểm tra.

- **Kiểm tra (Test):** Các bài kiểm tra tự động sẽ được chạy, bao gồm:
 - **Unit Test:** Kiểm tra các phần nhỏ nhất của mã nguồn để đảm bảo tính đúng đắn.
 - **Integration Test:** Kiểm tra xem các module hoặc các hệ thống có tương tác đúng như mong đợi không.
 - **Security Testing (Kiểm tra bảo mật):** Đây là bước quan trọng trong DevSecOps. Các công cụ kiểm tra bảo mật tự động (như **OWASP ZAP**, **SonarQube**, **Snyk**) sẽ được tích hợp vào quy trình CI để phát hiện các lỗ hổng bảo mật trong mã nguồn, chẳng hạn như SQL Injection, Cross-Site Scripting (XSS), và các lỗ hổng bảo mật khác. Nếu có lỗi bảo mật, hệ thống sẽ dừng và thông báo cho lập trình viên.
 - **Static Application Security Testing (SAST):** Phân tích mã nguồn tĩnh để tìm ra các vấn đề bảo mật tiềm ẩn trước khi phần mềm được chạy.
 - **Dynamic Application Security Testing (DAST):** Thực hiện kiểm tra bảo mật trên ứng dụng đang chạy, tìm kiếm lỗ hổng trong thời gian thực.

3. Continuous Delivery (CD):

- **Triển khai (Deploy):** Sau khi mọi bài kiểm tra (bao gồm cả kiểm tra bảo mật) thành công, ứng dụng sẽ được triển khai lên môi trường thử nghiệm (staging) hoặc môi trường gần giống với sản xuất để kiểm tra lại toàn bộ tính năng trước khi triển khai chính thức lên môi trường sản xuất.

4. Production Deployment:

- Nếu mọi thứ đều ổn định trong môi trường thử nghiệm, phần mềm sẽ được triển khai lên môi trường sản xuất (production). Quy trình này có thể là tự động hoàn toàn (Continuous Deployment), hoặc yêu cầu sự phê duyệt của người quản lý hệ thống trước khi thực hiện triển khai.

5. Giám sát và Phản hồi:

- **Giám sát ứng dụng:** Sau khi phần mềm được triển khai lên môi trường sản xuất, việc giám sát là vô cùng quan trọng để theo dõi hiệu suất và đảm bảo không có lỗi xảy ra. Các công cụ như **Prometheus**, **Grafana**, **New Relic**, hay **Datadog** sẽ giúp giám sát các chỉ số quan trọng như thời gian phản hồi, tài nguyên sử dụng, và các lỗi hệ thống.

- **Phản hồi thông qua kênh thông báo:** Nếu phát hiện lỗi hoặc vấn đề trong quá trình vận hành, hệ thống CI/CD sẽ gửi thông báo tới các thành viên trong đội ngũ phát triển hoặc quản trị qua các kênh như:
 - **Email:** Gửi thông báo lỗi qua email cho lập trình viên hoặc quản trị viên.
 - **Slack:** Gửi thông báo trực tiếp đến kênh Slack của nhóm phát triển để thông báo về sự cố hoặc kết quả kiểm tra.
 - **Telegram:** Gửi thông báo qua Telegram bot để đội ngũ nhận được phản hồi ngay lập tức trên ứng dụng di động.
 - **SMS:** Đối với các sự cố quan trọng, thông báo có thể được gửi qua SMS để đảm bảo rằng người nhận không bỏ sót thông tin.

3.2. Mối liên hệ giữa DevOps và CI/CD: DevOps thúc đẩy CI/CD, CI/CD thực thi DevOps

DevOps là phương pháp phát triển phần mềm kết hợp giữa phát triển (Development) và vận hành (Operations), giúp cải thiện sự hợp tác giữa các đội ngũ phát triển và vận hành, từ đó thúc đẩy việc phát hành phần mềm nhanh chóng và ổn định.

CI/CD là công cụ và quy trình giúp thực hiện triết lý của DevOps. **Continuous Integration (CI)** giúp tích hợp mã nguồn liên tục, trong khi **Continuous Delivery (CD)** giúp triển khai phần mềm một cách tự động và nhanh chóng.

Do đó, mối liên hệ giữa DevOps và CI/CD rất chặt chẽ:

- **DevOps thúc đẩy CI/CD:** Phương pháp DevOps yêu cầu tự động hóa các công đoạn trong quy trình phát triển phần mềm để giảm thiểu sai sót và tăng tốc độ phát triển. CI/CD chính là công cụ tự động hóa giúp thực hiện điều này.
- **CI/CD thực thi DevOps:** CI/CD là cách mà DevOps được triển khai vào thực tế. Nếu không có quy trình CI/CD, việc tích hợp mã nguồn và triển khai phần mềm sẽ gặp khó khăn, gây cản trở trong việc duy trì tốc độ phát triển và chất lượng phần mềm.

3.3. Các công cụ hỗ trợ CI/CD: Jenkins, GitLab CI, CircleCI, Travis CI, v.v.

Có rất nhiều công cụ hỗ trợ việc triển khai CI/CD, mỗi công cụ đều có những tính năng riêng biệt giúp các đội ngũ phát triển có thể tự động hóa quá trình kiểm tra, xây dựng và triển khai phần mềm.

1. Jenkins:

- **Jenkins** là một công cụ mã nguồn mở nổi tiếng trong việc thực hiện CI/CD. Nó giúp tự động hóa tất cả các giai đoạn trong quy trình phát triển phần mềm, từ việc xây dựng mã nguồn, kiểm tra tự động cho đến triển khai phần mềm.
- Jenkins có thể tích hợp với rất nhiều công cụ và plugin khác nhau, và đặc biệt phù hợp với các tổ chức lớn với nhu cầu tùy chỉnh cao.

2. **GitLab CI:**

- **GitLab CI** là một công cụ CI/CD tích hợp sẵn trong GitLab, một hệ thống quản lý mã nguồn Git. GitLab CI giúp tự động hóa quá trình từ commit mã nguồn, kiểm tra, xây dựng và triển khai ứng dụng.
- GitLab CI dễ sử dụng và không cần cài đặt thêm nhiều phần mềm ngoài GitLab. GitLab còn cung cấp các tính năng quản lý dự án, theo dõi bug, và các công cụ cộng tác, giúp tích hợp tất cả các bước phát triển trong một môi trường duy nhất.

3. **CircleCI:**

- **CircleCI** là một công cụ CI/CD mạnh mẽ, hỗ trợ việc tự động hóa quy trình phát triển phần mềm với khả năng tích hợp với nhiều hệ thống khác như GitHub, Bitbucket, Docker, Kubernetes. CircleCI cung cấp khả năng chạy các pipeline song song, giúp tiết kiệm thời gian khi kiểm tra và triển khai phần mềm.

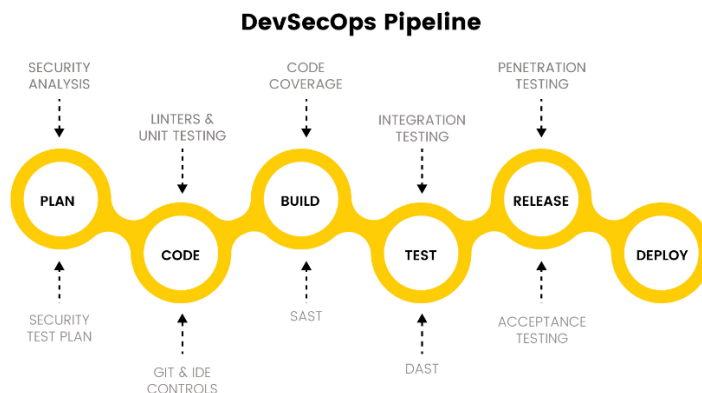
4. **Travis CI:**

- **Travis CI** là một công cụ CI/CD dễ sử dụng, phổ biến trong cộng đồng mã nguồn mở. Nó hỗ trợ nhiều ngôn ngữ lập trình và tích hợp tốt với GitHub. Travis CI giúp tự động hóa các công đoạn kiểm tra, xây dựng và triển khai ứng dụng, mang lại hiệu quả cao trong việc triển khai phần mềm.

5. **Các công cụ khác:**

- Ngoài các công cụ phổ biến trên, còn rất nhiều công cụ CI/CD khác như **TeamCity**, **Bamboo**, **GitHub Actions**, **Azure DevOps**, v.v. Tùy thuộc vào yêu cầu của dự án và môi trường phát triển, các đội ngũ phát triển sẽ chọn công cụ phù hợp nhất.

4. Các loại test trong DevSecOps



4.1. Static Application Security Testing (SAST) – Kiểm tra mã nguồn để phát hiện lỗ hổng bảo mật

SAST là một phương pháp kiểm tra bảo mật phần mềm ngay khi ứng dụng đang được viết hoặc phát triển. SAST sẽ quét mã nguồn của ứng dụng để tìm ra các lỗ hổng bảo mật tiềm ẩn, chẳng hạn như **SQL Injection**, **Cross-Site Scripting (XSS)**, **Buffer Overflow**, v.v. Phương pháp này giúp phát hiện các vấn đề bảo mật từ rất sớm trong quá trình phát triển, trước khi ứng dụng được biên dịch và chạy.

- **Cách thức hoạt động:** Công cụ SAST phân tích mã nguồn hoặc bytecode của ứng dụng mà không cần chạy ứng dụng. Các công cụ như **SonarQube**, **Checkmarx**, **Fortify** có thể quét toàn bộ mã nguồn, giúp tìm ra các lỗ hổng tiềm ẩn trong ứng dụng.
- **Lợi ích:** Phát hiện lỗ hổng bảo mật ngay từ giai đoạn phát triển, giúp tiết kiệm thời gian và chi phí sửa chữa sau này.

4.2. Software Composition Analysis (SCA) – Kiểm tra các thành phần bên ngoài (dependencies) của ứng dụng

SCA là quá trình kiểm tra các **thư viện**, **frameworks**, và **dependencies** mà ứng dụng sử dụng. Các thành phần này có thể chứa các lỗ hổng bảo mật nếu chúng không được cập nhật thường xuyên. Công cụ SCA sẽ giúp phát hiện các thư viện có lỗ hổng bảo mật đã được công nhận và cung cấp bản vá.

- **Cách thức hoạt động:** SCA quét tất cả các thư viện bên ngoài mà ứng dụng đang sử dụng (ví dụ: **npm**, **Maven**, **pip**), so sánh với cơ sở dữ liệu các lỗ hổng bảo mật đã biết (chẳng hạn như **CVE** – Common Vulnerabilities and Exposures).
- **Lợi ích:** Đảm bảo rằng các thành phần bên ngoài không có lỗ hổng bảo mật, giảm nguy cơ bị tấn công từ các thư viện hoặc framework cũ không được cập nhật.

4.3. Image Scan – Kiểm tra bảo mật container image

Trong môi trường DevSecOps, **containerization** là một phương pháp phổ biến để triển khai ứng dụng. Tuy nhiên, các container image có thể chứa các lỗ hổng bảo mật nếu không được kiểm tra cẩn thận trước khi sử dụng. **Image Scan** giúp kiểm tra các image Docker hoặc container khác để phát hiện các lỗ hổng bảo mật trong quá trình xây dựng và triển khai ứng dụng.

- **Cách thức hoạt động:** Các công cụ như **Clair**, **Trivy**, hoặc **Anchore** sẽ quét các container image để tìm kiếm các lỗ hổng bảo mật trong phần mềm mà image chứa, chẳng hạn như các lỗ hổng hệ điều hành hoặc thư viện có vấn đề.
- **Lợi ích:** Đảm bảo các container image không chứa các vấn đề bảo mật trước khi triển khai lên môi trường sản xuất.

4.4. Dynamic Application Security Testing (DAST) – Kiểm tra ứng dụng trong runtime (thực thi)

DAST là một phương pháp kiểm tra bảo mật ứng dụng khi ứng dụng đang chạy. DAST thực hiện các kiểm tra bảo mật **nội dung động** trong khi ứng dụng đang thực thi (runtime), giống như cách mà kẻ tấn công có thể thử nghiệm các lỗ hổng trên ứng dụng.

- **Cách thức hoạt động:** DAST không yêu cầu truy cập mã nguồn của ứng dụng. Nó kiểm tra các lỗ hổng bảo mật trong giao diện người dùng, API, và các giao tiếp giữa các thành phần của ứng dụng khi đang hoạt động. Các công cụ phổ biến như **OWASP ZAP** hoặc **Burp Suite** được sử dụng để quét và kiểm tra các lỗ hổng như **Cross-Site Scripting (XSS)**, **SQL Injection**, **Command Injection**, v.v.
- **Lợi ích:** DAST giúp phát hiện lỗ hổng bảo mật mà SAST không thể phát hiện được, vì nó kiểm tra ứng dụng trong môi trường thực tế.

4.5. Performance Testing – Kiểm tra hiệu năng của ứng dụng

Performance Testing kiểm tra khả năng của ứng dụng trong việc xử lý các tác vụ với hiệu suất cao dưới các điều kiện tải khác nhau. Các bài kiểm tra hiệu năng giúp đảm bảo ứng dụng hoạt động ổn định và không bị gián đoạn khi lượng người dùng hoặc dữ liệu tăng lên.

- **Cách thức hoạt động:** Các công cụ như **JMeter**, **Gatling**, **LoadRunner** được sử dụng để mô phỏng một số lượng lớn người dùng truy cập ứng dụng cùng lúc. Các chỉ số như **response time**, **throughput**, và **resource usage** sẽ được đo và đánh giá.
- **Lợi ích:** Đảm bảo ứng dụng có thể chịu tải cao mà không gặp phải tình trạng giảm hiệu suất hoặc sự cố hệ thống khi ứng dụng lên môi trường sản xuất.

4.6. Other tests: Unit Tests, Integration Tests, Acceptance Tests

Ngoài các kiểm tra bảo mật và hiệu suất, còn có những loại kiểm tra cơ bản khác trong quy trình phát triển phần mềm, bao gồm:

- **Unit Tests:** Kiểm tra từng phần nhỏ của mã nguồn (unit) để đảm bảo mỗi phần hoạt động đúng như mong đợi. Đây là các bài kiểm tra cơ bản trong quy trình phát triển phần mềm.
- **Integration Tests:** Kiểm tra tính tương thích và sự kết hợp giữa các module hoặc thành phần khác nhau của ứng dụng.
- **Acceptance Tests:** Kiểm tra ứng dụng từ góc độ người dùng cuối để đảm bảo phần mềm đáp ứng các yêu cầu kinh doanh và chức năng của người dùng.

III. Các công cụ và công nghệ sử dụng trong quy trình

1. Docker

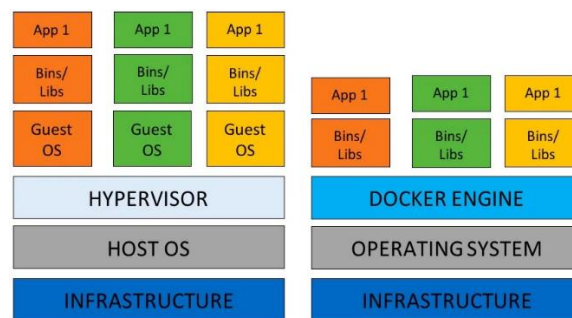
1.1. Khái niệm về Docker và containerization

Docker là một nền tảng giúp chúng ta xây dựng, vận chuyển và chạy các ứng dụng trong môi trường chứa gọi là container. Container là một loại công nghệ ảo hóa nhẹ giúp đóng gói ứng dụng và các phụ thuộc của nó (như thư viện, cấu hình hệ thống, v.v.) vào trong một gói duy nhất. Mỗi container hoạt động như một môi trường cô lập, giúp ứng dụng chạy được một cách nhất quán trên mọi hệ thống mà không bị ảnh hưởng bởi các sự khác biệt trong môi trường của host.



Docker giúp tách biệt ứng dụng và môi trường chạy của nó, giúp chúng ta có thể dễ dàng triển khai, mở rộng và vận hành ứng dụng mà không gặp phải vấn đề "works on my machine". Điều này rất quan trọng trong việc phát triển và triển khai phần mềm, đặc biệt khi sử dụng CI/CD để tự động hóa các quy trình này.

Containerization (hay còn gọi là Container hóa) là quá trình đóng gói các ứng dụng vào trong các container. Điều này giúp ứng dụng chạy ổn định và nhất quán bất kể môi trường phát triển, kiểm thử hay sản xuất.



Một điểm quan trọng là, so với máy ảo (Virtual Machine), Docker sử dụng các container nhẹ hơn và khởi động nhanh hơn rất nhiều, vì chúng không cần phải có một hệ điều hành riêng biệt mà chia sẻ phần nhân (kernel) của hệ điều hành chủ.

1.2. Tối ưu vào bảo mật Docker Image

Khi xây dựng Docker image, việc tối ưu hóa Dockerfile là rất quan trọng để giảm kích thước image và tăng hiệu suất.

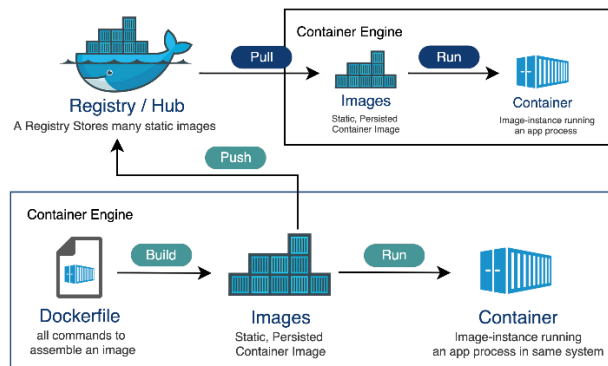
- **Sử dụng image nhỏ:** Chọn các base image nhỏ, chẳng hạn như alpine thay vì các image lớn như ubuntu. Điều này giúp giảm kích thước của container.
- **Giảm số lượng layer:** Mỗi lệnh trong Dockerfile (như RUN, COPY, ADD) tạo ra một layer mới trong Docker image. Cố gắng gộp các lệnh lại với nhau để giảm số lượng layer.
- **Sử dụng cache layer hiệu quả:** Docker cache các layer đã được xây dựng trước đó, giúp tăng tốc quá trình build nếu không có thay đổi. Bạn có thể tận dụng điều này bằng cách sắp xếp các lệnh Dockerfile sao cho ít thay đổi nhất có thể.

Bên cạnh đó, bảo mật là một phần quan trọng khi sử dụng Docker, vì container có thể bị khai thác nếu không được quản lý đúng cách.

1. **Cập nhật thường xuyên:** Đảm bảo các Docker image được cập nhật thường xuyên để vá các lỗ hổng bảo mật.
2. **Chạy container với quyền hạn thấp:** Tránh chạy container với quyền root. Sử dụng các người dùng có quyền hạn thấp trong container để giảm thiểu các nguy cơ bảo mật.
3. **Quản lý Docker secrets:** Docker cung cấp cách thức để lưu trữ các thông tin nhạy cảm như mật khẩu, API key một cách an toàn thông qua Docker secrets.

1.3. Lưu trữ Docker Image trong Quy trình CI/CD

Khi xây dựng Docker image trong quy trình CI/CD, sẽ cần có một nơi để lưu trữ và phân phối các image này. Docker images là những tệp tin nhị phân (binary) chứa mã nguồn và các dependency của ứng dụng, cho phép ứng dụng chạy trong một container. Để chia sẻ Docker image giữa các môi trường (development, staging, production), chúng ta cần phải sử dụng một registry, tức là nơi lưu trữ và quản lý các Docker image.



Docker cung cấp một registry public miễn phí là Docker Hub, ngoài ra, cũng có thể sử dụng các registry khác như JFrog Artifactory, Harbor, Google Container Registry (GCR), Amazon Elastic Container Registry (ECR):

- Docker Hub: Registry công cộng miễn phí, dễ sử dụng. Có thể tải lên và chia sẻ Docker image công khai hoặc riêng tư.
- JFrog Artifactory: Công cụ mạnh mẽ cho quản lý Docker image, bảo mật tốt và hỗ trợ nhiều loại artifact.
- Harbor: Registry mã nguồn mở, cung cấp các tính năng bảo mật như quét lỗ hổng và quản lý quyền truy cập.
- GCR (Google Container Registry), ECR (AWS Elastic Container Registry): Dịch vụ đám mây của Google và AWS, tích hợp tốt với các dịch vụ container của họ.

Lợi ích của Docker Registry trong CI/CD:

- Tự động hóa: Quá trình build, push, và pull Docker image có thể tự động hóa trong pipeline CI/CD, tiết kiệm thời gian và công sức.
- Quản lý phiên bản: Docker Registry giúp theo dõi và quản lý các phiên bản image dễ dàng, hỗ trợ rollback và upgrade ứng dụng nhanh chóng.
- Bảo mật: Các registry như Artifactory và Harbor cung cấp các tính năng bảo mật như kiểm soát quyền truy cập và quét lỗ hổng bảo mật.

1.4. Docker trong quy trình CI/CD

Trong quy trình CI/CD, Docker mang lại nhiều lợi ích lớn:

1. **Nhất quán trong môi trường phát triển và sản xuất:** Docker giúp bạn đảm bảo rằng môi trường chạy ứng dụng trong quá trình phát triển và sản xuất hoàn toàn giống nhau. Việc chạy ứng dụng trong container trên máy phát triển của lập trình viên cũng giống như khi chạy trên máy chủ production.
2. **Khả năng tái sử dụng và chia sẻ:** Bạn có thể đóng gói ứng dụng của mình vào Docker image và chia sẻ cho các thành viên khác trong nhóm mà không lo lắng về vấn đề tương thích hệ điều hành hay phần mềm cài sẵn.
3. **Tiết kiệm tài nguyên:** So với việc sử dụng máy ảo, Docker chỉ cần chia sẻ hệ điều hành với máy chủ chủ, vì thế nó ít tốn tài nguyên và khởi động nhanh hơn rất nhiều.
4. **Tự động hóa dễ dàng:** Docker dễ dàng tích hợp vào các công cụ CI/CD như GitLab CI/CD để tự động hóa quá trình xây dựng, kiểm thử và triển khai ứng dụng.
5. **Dễ dàng scale (mở rộng):** Với Docker, việc mở rộng ứng dụng để chạy trên nhiều máy chủ hay dịch vụ container orchestration như Kubernetes trở nên dễ dàng hơn nhiều.

Docker thường được tích hợp vào quy trình CI/CD để tự động hóa các bước từ xây dựng, kiểm thử đến triển khai ứng dụng.

1. Xây dựng Docker image:

Mỗi ứng dụng cần có một Docker image riêng, để có thể đóng gói và triển khai ứng dụng đó vào bất kỳ môi trường nào. Docker image có thể được xây dựng từ một file cấu hình gọi là Dockerfile. File này chứa các chỉ thị để Docker biết cách xây dựng image từ các bước như cài đặt phần mềm, sao chép mã nguồn vào container, v.v.

```
FROM node:14

WORKDIR /app

COPY package.json /app

RUN npm install

COPY . /app

CMD ["npm", "start"]
```

2. Cấu hình CI/CD pipeline với Docker:

Trong GitLab CI/CD, bạn có thể tích hợp Docker vào quy trình tự động của mình. GitLab CI sử dụng file `.gitlab-ci.yml` để cấu hình các bước như build, test và deploy ứng dụng.

```
stages:
  - build
  - test
  - push
  - deploy

build:
  stage: build
  script:
    - docker build -t my-app .

test:
  stage: test
  script:
    - docker run my-app npm test

push:
  stage: push
  script:
    - docker push repository/my-app:tag

deploy:
  stage: deploy
  script:
    - docker run -d my-app
```

2. GitLab và GitLab CI

2.1. Gitlab

GitLab là một nền tảng DevOps tích hợp toàn bộ, cung cấp các công cụ giúp phát triển phần mềm từ giai đoạn lập kế hoạch, phát triển mã nguồn, kiểm thử, triển khai đến bảo trì. GitLab giúp các nhóm phát triển phần mềm làm việc hiệu quả hơn thông qua các tính năng như quản lý mã nguồn (Git), theo dõi vấn đề, CI/CD, và nhiều công cụ hỗ trợ khác.

- **GitLab** là một hệ thống quản lý mã nguồn phân tán dựa trên Git. Từ GitLab, các lập trình viên có thể dễ dàng quản lý mã nguồn, cộng tác với nhau và theo dõi các thay đổi.
- Ngoài việc là một hệ thống quản lý mã nguồn, GitLab còn cung cấp rất nhiều tính năng hỗ trợ cho quy trình DevOps, đặc biệt là **GitLab CI/CD**, cho phép tự động hoá các bước như kiểm thử, xây dựng, triển khai, và giám sát.

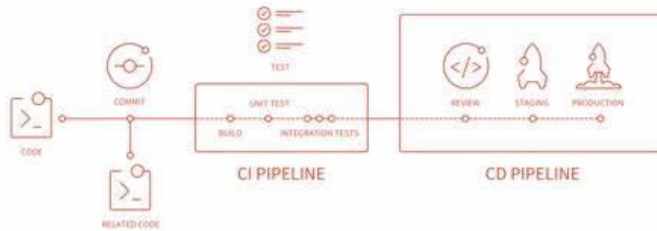
GitLab hỗ trợ **DevOps** và **CI/CD** bằng cách tích hợp tất cả các công cụ trong một nền tảng duy nhất, giúp các nhóm phát triển phần mềm có thể làm việc hiệu quả và liên tục. Quy trình CI/CD (Continuous Integration và Continuous Deployment) giúp tự động hóa việc kiểm tra mã nguồn, xây dựng ứng dụng, và triển khai sản phẩm lên môi trường sản xuất.

- **Continuous Integration (CI)**: Quá trình liên tục tích hợp mã nguồn vào một repository chung. Mỗi lần có thay đổi mới từ các lập trình viên, GitLab CI sẽ tự động kiểm tra và xây dựng ứng dụng.
- **Continuous Delivery/Deployment (CD)**: Sau khi mã được tích hợp, ứng dụng sẽ được tự động triển khai lên môi trường kiểm thử, staging và thậm chí cả môi trường sản xuất mà không cần sự can thiệp thủ công.

GitLab giúp cho việc xây dựng, kiểm tra và triển khai ứng dụng trở nên nhanh chóng, dễ dàng và giảm thiểu lỗi do thao tác thủ công.

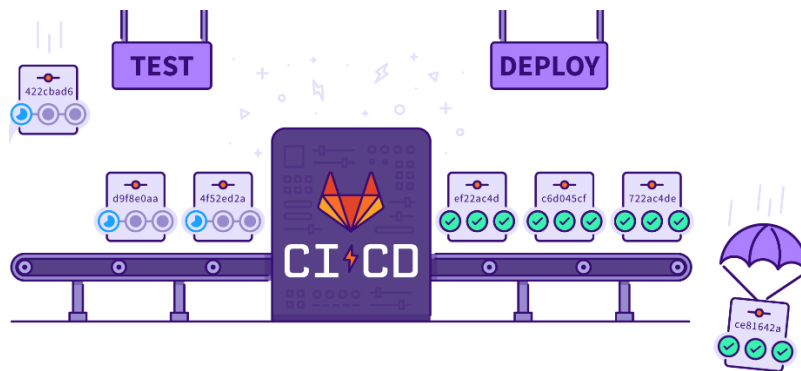
2.2. GitLab CI

GitLab CI (Continuous Integration) là **một hệ thống tự động hoá quy CI/CD trong GitLab**. GitLab CI giúp tự động hoá quá trình kiểm tra, xây dựng, và triển khai phần mềm mỗi khi có thay đổi mã nguồn. GitLab CI cho phép lập trình viên nhanh chóng phát hiện lỗi trong mã nguồn, giảm thiểu các bước thủ công và nâng cao chất lượng sản phẩm phần mềm.



GitLab CI được tích hợp trực tiếp trong GitLab, giúp việc quản lý mã nguồn, theo dõi vấn đề, và CI/CD trở nên dễ dàng hơn. Khi bạn đẩy mã (commit) lên GitLab, GitLab CI sẽ tự động bắt đầu chạy các pipeline, bao gồm các công việc (jobs) như kiểm thử, xây dựng phần mềm, và triển khai sản phẩm lên môi trường thử nghiệm hoặc sản xuất.

Tiếp theo chúng ta sẽ tìm hiểu về các thành phần trong Gitlab CI. Đầu tiên, **Pipeline** là một tập hợp các công việc tự động hoá trong GitLab CI, giúp thực hiện các tác vụ như kiểm thử, biên dịch, triển khai, v.v. Mỗi pipeline có thể bao gồm nhiều jobs và được tổ chức thành các stages (giai đoạn). Pipeline sẽ tự động chạy mỗi khi có sự thay đổi trong mã nguồn.



Cấu trúc cơ bản của một Pipeline:

- Pipeline bắt đầu khi có một sự kiện, chẳng hạn như một commit được đẩy lên GitLab.
- Mỗi pipeline bao gồm một chuỗi các stages.
- Mỗi stage bao gồm một hoặc nhiều jobs.

```

stages:
- build
- test
- deploy

build-job:
stage: build
script:
- echo "Building application..."

test-job:
stage: test
script:
- echo "Running tests..."

deploy-job:
stage: deploy
script:
- echo "Deploying to production..."

```

Tiếp theo, **GitLab Runner**, là công cụ thực thi các jobs trong pipeline. Khi cấu hình một pipeline trong GitLab CI, GitLab sẽ gửi các jobs tới runner để thực thi các lệnh trong đó. Runners có thể là môi trường máy chủ hoặc máy tính có thể chạy các tác vụ của pipeline.



Các loại GitLab Runner:

- **Shared Runners:** Là những runners mà GitLab cung cấp sẵn và có thể dùng chung cho tất cả các dự án. Chúng thường được cấu hình sẵn trong GitLab CI và bạn không cần cài đặt gì thêm.
- **Group Runners:** Là những runners mà GitLab cung cấp sẵn và có thể dùng chung cho tất cả các dự án trong một group cụ thể. Những runner này có thể được cấu hình và cài đặt trên máy chủ của bạn hoặc trên các môi trường của riêng bạn.
- **Specific Runners:** Giống với Group Runners nhưng chỉ được gán cho một dự án cụ thể.

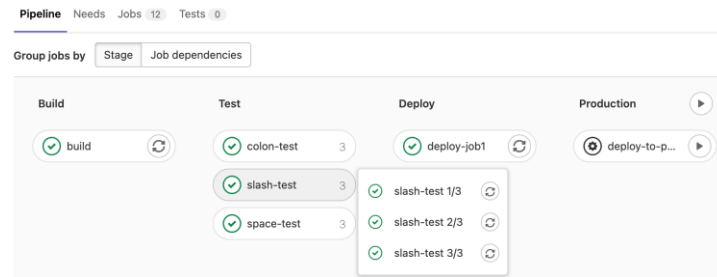
Các Runner hoạt động như thế nào?

- Mỗi khi có một pipeline mới cần thực thi, GitLab sẽ giao các jobs trong pipeline cho một hoặc nhiều runners.
- Runners có thể chạy các jobs trên các môi trường khác nhau như máy chủ ảo, Docker container, hoặc thậm chí trên máy tính cá nhân.

Ví dụ:

- Khi chạy một pipeline trên GitLab CI, các jobs trong pipeline sẽ được gửi đến một runner.
- Runner này sẽ thực thi các lệnh trong script của mỗi job (ví dụ: npm install, npm test, v.v.).

Jobs trong GitLab CI là các tác vụ cụ thể mà pipeline sẽ thực thi. Mỗi job có một nhiệm vụ rõ ràng, chẳng hạn như kiểm tra mã nguồn, biên dịch ứng dụng, chạy kiểm thử tự động, triển khai phần mềm lên môi trường thử nghiệm hoặc sản xuất, v.v.



Cấu hình một job trong pipeline:

- Mỗi job trong GitLab CI được định nghĩa bằng tên job và các lệnh cần chạy.
- Mỗi job có thể thuộc về một stage nhất định, ví dụ: build, test, deploy.

```

build:
  stage: build
  script:
    - echo "Building the project..."
    - npm install

```

Trong ví dụ trên, job build sẽ được thực thi trong **stage** build và chạy lệnh npm install để cài đặt các phụ thuộc của dự án.

Một job có thể có các thuộc tính như:

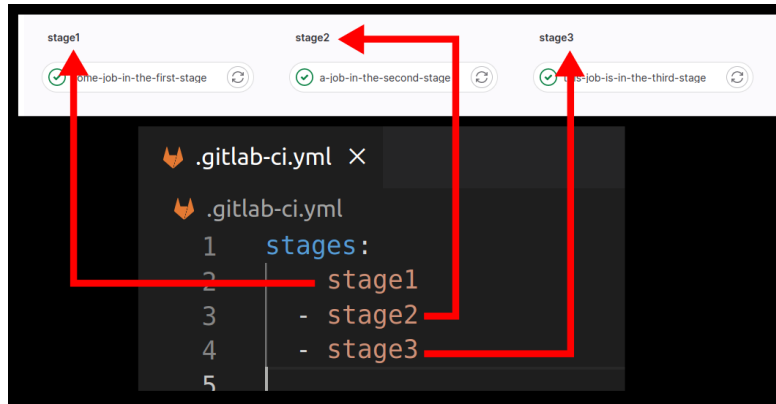
- script: Các lệnh sẽ được thực thi trong job này.
- stage: Giai đoạn mà job này thuộc về (build, test, deploy, v.v.).
- only: Điều kiện để job này chỉ chạy trên các nhánh hoặc môi trường cụ thể.
- before_script và after_script: Các lệnh thực thi trước và sau khi job hoàn thành.
- Ngoài ra còn rất nhiều thuộc tính khác để cấu hình cho 1 job

Lưu ý khi cấu hình Jobs:

- Mỗi job có thể chạy trên một runner khác nhau, và các jobs có thể chạy song song hoặc tuần tự tùy thuộc vào cách bạn cấu hình stages.

- Jobs giúp tự động hoá các bước lặp đi lặp lại trong quy trình phát triển phần mềm.

Stages là các bước lớn trong pipeline, giúp phân loại các công việc cần thực hiện. Một pipeline có thể có nhiều stages (ví dụ: build, test, deploy), và các jobs sẽ được thực hiện theo từng stage.



Cấu hình stages:

- Các stages giúp phân chia các jobs thành từng giai đoạn cụ thể, mỗi giai đoạn có các công việc liên quan như xây dựng, kiểm thử, hoặc triển khai.
- Stages trong GitLab CI được định nghĩa trong file .gitlab-ci.yml và sẽ thực thi lần lượt theo thứ tự khai báo.

Lợi ích của việc sử dụng stages:

- Stages giúp quản lý quá trình CI/CD dễ dàng hơn bằng cách chia quy trình thành các bước rõ ràng.
- Giảm thiểu lỗi khi phát hiện lỗi sớm trong các giai đoạn đầu của pipeline, chẳng hạn như trong giai đoạn kiểm thử.

Để tạo và cấu hình một pipeline cơ bản trên GitLab CI, chỉ cần làm theo các bước sau:

1. Tạo file .gitlab-ci.yml:

- File này phải nằm trong thư mục gốc của dự án GitLab.
- Nội dung của file sẽ khai báo các stages, jobs, và runners để GitLab CI biết phải làm gì..

2. Đẩy file lên GitLab:

- Sau khi tạo xong file .gitlab-ci.yml, bạn commit và push file này lên GitLab.

3. Chạy pipeline:

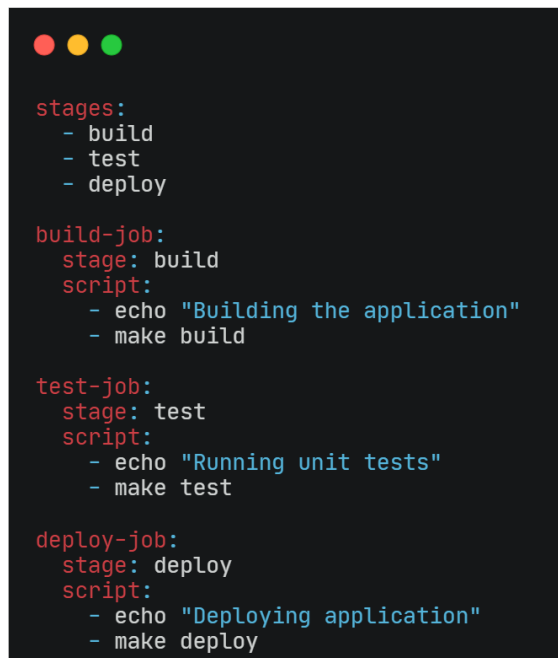
- Mỗi lần push code lên GitLab, pipeline sẽ tự động chạy.

- Chúng ta có thể theo dõi tiến trình và kết quả của pipeline trong tab "CI/CD" của GitLab.

Một pipeline cơ bản có thể bao gồm các bước như sau:

1. **Build:** Xây dựng mã nguồn thành một ứng dụng có thể chạy được.
2. **Test:** Kiểm tra mã nguồn để đảm bảo không có lỗi trong ứng dụng.
3. **Deploy:** Triển khai ứng dụng lên môi trường thử nghiệm (staging) hoặc sản xuất (production).
4. Ngoài ra còn có thể có các bước test như unit test, SAST, DAST, vv. Có thể tích hợp thêm việc thông báo kết quả file report về email, slack, telegram, ...

Ví dụ về một GitLab CI pipeline đơn giản để xây dựng và triển khai ứng dụng:



```
stages:
  - build
  - test
  - deploy

build-job:
  stage: build
  script:
    - echo "Building the application"
    - make build

test-job:
  stage: test
  script:
    - echo "Running unit tests"
    - make test

deploy-job:
  stage: deploy
  script:
    - echo "Deploying application"
    - make deploy
```

Trong ví dụ trên:

- **stages:** Khai báo các stages cho pipeline, bao gồm các giai đoạn build, test, và deploy.
- **jobs:** Mỗi job tương ứng với một giai đoạn. Ví dụ, build-job sẽ chạy trong stage build, test-job trong stage test, và deploy-job trong stage deploy.
- **script:** Phần này chứa các lệnh mà bạn muốn GitLab CI thực hiện trong mỗi job. Chẳng hạn, make build, make test, và make deploy là các lệnh giả định cho việc xây dựng, kiểm tra và triển khai ứng dụng.

Sau khi đẩy code lên GitLab, pipeline sẽ tự động được kích hoạt và chạy theo các stages mà đã được cấu hình. GitLab sẽ cung cấp giao diện để theo dõi tình trạng của các jobs trong pipeline, giúp bạn dễ dàng nhận ra các lỗi hoặc vấn đề.

3. Xây dựng Pipeline với GitLab CI và Docker

Trong phần này, chúng ta sẽ đi qua quy trình cấu hình và xây dựng một GitLab CI/CD pipeline với Docker. Mục tiêu là tự động hóa các bước từ việc xây dựng ứng dụng, kiểm thử cho đến triển khai trên môi trường sản xuất, đồng thời tích hợp các công cụ bảo mật vào pipeline để đảm bảo ứng dụng không chỉ chạy mượt mà mà còn an toàn.

3.1. Cấu hình GitLab CI để chạy các bước CI/CD với Docker

GitLab CI (Continuous Integration/Continuous Deployment) là một công cụ mạnh mẽ để tự động hóa quá trình phát triển phần mềm. Để GitLab CI có thể xây dựng và triển khai ứng dụng Docker, ta cần cấu hình file `.gitlab-ci.yml`.

File `.gitlab-ci.yml` là nơi xác định các job và pipeline cho GitLab CI. Mỗi job sẽ thực thi một tác vụ cụ thể như build, test, deploy ứng dụng. Để xây dựng một pipeline sử dụng Docker, trước hết ta cần tạo một file `.gitlab-ci.yml` cơ bản với nội dung như sau:

```
stages:
  - build
  - test
  - deploy

# Job build: Build docker image
build:
  stage: build
  script:
    - docker build -t my-app:$CI_COMMIT_REF_NAME .
  tags:
    - docker

# Job test: Test application
test:
  stage: test
  script:
    - docker run --rm my-app:$CI_COMMIT_REF_NAME npm test
  tags:
    - docker

# Job deploy: Deploy Docker container to production
deploy:
  stage: deploy
  script:
    - docker run -d -p 8080:8080 my-app:$CI_COMMIT_REF_NAME
  only:
    - master
  tags:
    - docker
```

Giải thích các phần chính trong cấu hình:

- **stages:** Định nghĩa các giai đoạn (stages) của pipeline. Trong ví dụ trên có 3 giai đoạn: build, test, và deploy.

- job build: Tạo Docker image từ file Dockerfile có sẵn trong repo. `docker build -t my-app:$CI_COMMIT_REF_NAME` . sẽ tạo một Docker image mới với tên my-app và tag theo tên nhánh hoặc commit ID.
- job test: Sau khi image được tạo, job này sẽ chạy thử ứng dụng trong container để kiểm tra xem ứng dụng có chạy đúng hay không.
- job deploy: Nếu các bước trước thành công, job deploy sẽ tự động triển khai Docker container lên môi trường sản xuất. Ở đây, chỉ có nhánh master mới được phép triển khai.

Các bước cơ bản khi cấu hình GitLab CI với Docker:

- Đảm bảo Docker đã được cài đặt trên runner của GitLab.
- Tạo file `.gitlab-ci.yml` trong thư mục gốc của dự án.
- Định nghĩa các job và các bước cần thiết cho pipeline.

3.2. Tạo các job trong GitLab CI để build, test, và deploy ứng dụng Docker

Sau khi cấu hình xong GitLab CI, chúng ta sẽ tạo các job để thực hiện các bước mà chúng ta mong muốn trong pipeline. Tùy thuộc vào từng dự án khác nhau sẽ thực hiện các bước với yêu cầu khác nhau. Sau đây là ví dụ về 1 số job có thể thường sử dụng:

1. Job Build: có nhiệm vụ tạo một Docker image từ mã nguồn của ứng dụng. Dockerfile là tệp cấu hình hướng dẫn Docker cách xây dựng ứng dụng. Mỗi khi có thay đổi trong mã nguồn, GitLab CI sẽ tự động xây dựng lại image để đảm bảo phiên bản mới nhất của ứng dụng.
2. Job Test: Sau khi Docker image được xây dựng, job Test sẽ kiểm tra ứng dụng trong môi trường Docker container. Đây là một bước quan trọng để đảm bảo rằng ứng dụng không gặp lỗi nghiêm trọng và hoạt động như mong đợi.
3. Job Deploy: Sau khi build và test thành công, job Deploy sẽ triển khai ứng dụng lên môi trường. Trong trường hợp này, chúng ta sẽ chạy Docker container trên máy chủ hoặc dịch vụ cloud như AWS, GCP, hoặc Azure.

3.3. Tích hợp các công cụ bảo mật vào pipeline (SAST, Image Scanning, DAST)

Để bảo vệ ứng dụng khỏi các mối đe dọa bảo mật, chúng ta cần tích hợp các công cụ bảo mật vào pipeline. Các công cụ bảo mật phổ biến bao gồm:

SAST (Static Application Security Testing): SAST là công cụ quét mã nguồn để tìm kiếm các lỗ hổng bảo mật trong mã ứng dụng. Bạn có thể tích hợp công cụ SAST vào GitLab CI bằng cách sử dụng GitLab's Security Dashboard hoặc các công cụ như SonarQube, Semgrep,...

```
sast:
  stage: test
  script:
    - docker run --rm -v $(pwd):/code securecodebox/sast bandit -r /code
```

Image Scanning: Quét Docker images để phát hiện các lỗ hổng bảo mật tiềm ẩn trong các dependency. GitLab cung cấp khả năng quét hình ảnh Docker thông qua container scanning. Bạn có thể tích hợp công cụ quét như Clair hoặc Trivy.

```
image_scanning:
  stage: test
  script:
    - docker run --rm -v $(pwd):/project anchore/anchore-cli image scan my-app:$CI_COMMIT_REF_NAME
```

DAST (Dynamic Application Security Testing): DAST quét ứng dụng khi nó đang chạy, tìm kiếm các lỗ hổng bảo mật trong quá trình thực thi. Công cụ như OWASP ZAP có thể được tích hợp vào pipeline để kiểm tra các ứng dụng web.

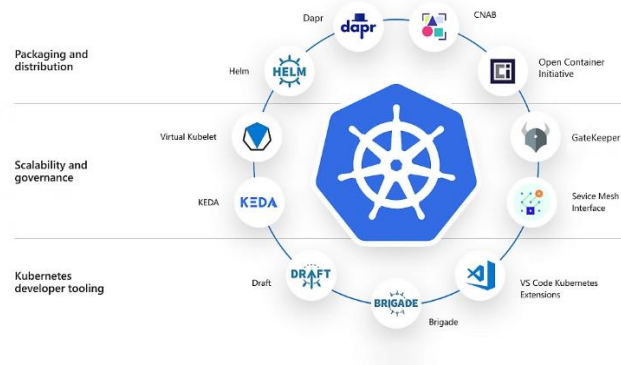
```
dast:
  stage: test
  script:
    - docker run --rm zaproxy/zaproxy zap-baseline.py -t http://localhost:8080
```

Việc xây dựng một CI/CD pipeline với GitLab và Docker giúp tự động hóa quy trình phát triển phần mềm, từ việc xây dựng ứng dụng, kiểm thử, đến triển khai. Bằng cách tích hợp các công cụ bảo mật như SAST, Image Scanning và DAST, chúng ta không chỉ đảm bảo chất lượng mã nguồn mà còn bảo vệ ứng dụng khỏi các nguy cơ bảo mật. Đó là một quy trình quan trọng trong DevSecOps giúp phát triển phần mềm an toàn và hiệu quả hơn.

4. Kubernetes

4.1. Tổng quan về Kubernetes và vai trò trong việc quản lý container

Kubernetes là gì? Kubernetes (thường gọi tắt là K8s) là một hệ thống mã nguồn mở dùng để tự động hoá việc triển khai, quản lý và mở rộng các ứng dụng container. Kubernetes được phát triển bởi Google và hiện nay được duy trì bởi Cloud Native Computing Foundation (CNCF).

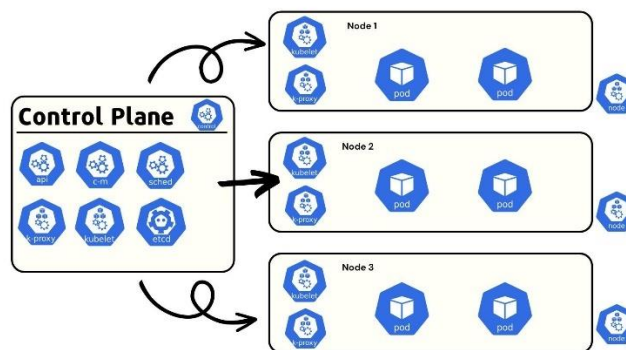


K8s giúp quản lý các container (ví dụ: Docker container) trong một môi trường phân tán, giúp chúng ta dễ dàng triển khai, mở rộng và quản lý các ứng dụng với độ tin cậy cao.

Lợi ích khi sử dụng Kubernetes cho production environment:

- **Tự động mở rộng:** Kubernetes có khả năng tự động scale (mở rộng) hoặc thu nhỏ các ứng dụng dựa trên nhu cầu, giúp tiết kiệm tài nguyên và chi phí.
- **Khả năng chịu lỗi:** Kubernetes đảm bảo rằng nếu có một container hoặc node gặp sự cố, nó sẽ tự động khởi động lại hoặc thay thế các instance bị lỗi.
- **Quản lý tài nguyên hiệu quả:** Kubernetes giúp phân bổ tài nguyên (CPU, bộ nhớ) cho các container một cách hợp lý.
- **Tự động cập nhật và rollback:** K8s hỗ trợ triển khai các phiên bản mới của ứng dụng (rolling update) mà không làm gián đoạn dịch vụ. Nếu có vấn đề, Kubernetes có thể tự động quay lại phiên bản trước (rollback).

Kubernetes giúp chúng ta quản lý container trên một cụm máy tính (cluster). Trong một cluster Kubernetes, bạn có thể triển khai các ứng dụng phức tạp, quản lý số lượng lớn container, đảm bảo ứng dụng hoạt động một cách ổn định và có thể mở rộng linh hoạt.



K8s giúp tổ chức các container vào các đơn vị gọi là **Pods** và điều phối chúng một cách tự động.

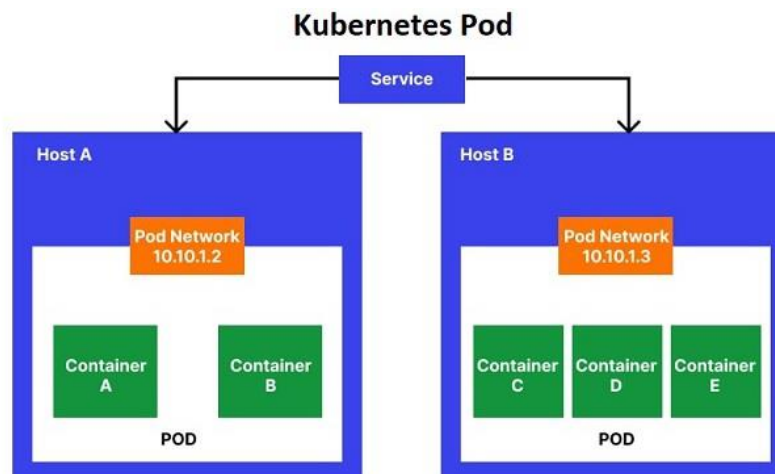
- **Cluster:** Là môi trường Kubernetes bao gồm một hoặc nhiều máy chủ (node), được chia thành 2 loại: master node (quản lý và điều phối) và worker node (chạy các ứng dụng).
- **Pod:** Là đơn vị cơ bản trong Kubernetes, chứa một hoặc nhiều container, và chia sẻ tài nguyên như mạng và bộ nhớ.
- **Node:** Là một máy chủ vật lý hoặc ảo, có thể là máy chủ trong datacenter hoặc cloud, nơi các container và pods được triển khai.

4.2. Các thành phần chính của Kubernetes

Kubernetes có rất nhiều thành phần, nhưng dưới đây là một số thành phần cơ bản cần nắm vững khi làm việc với Kubernetes:

1. Pod

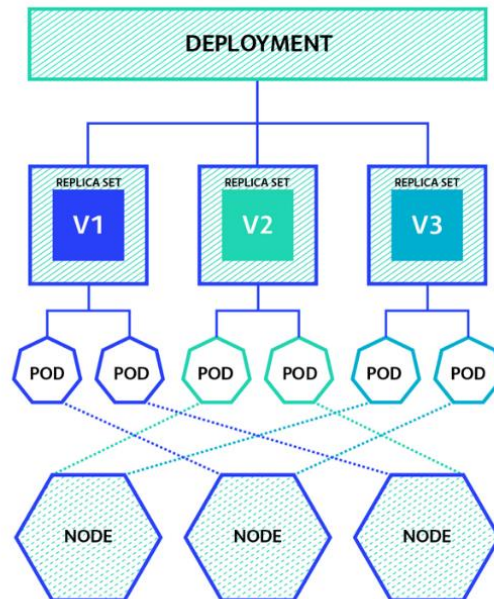
- Pod là đơn vị cơ bản nhất trong Kubernetes, đại diện cho một hoặc nhiều container được chạy cùng nhau trên một máy chủ. Các container trong cùng một pod chia sẻ mạng, storage và có thể giao tiếp với nhau một cách trực tiếp.
- Một pod có thể chứa một hoặc nhiều container, và các container trong một pod thường làm việc với nhau để thực hiện một chức năng chung.



2. Deployment

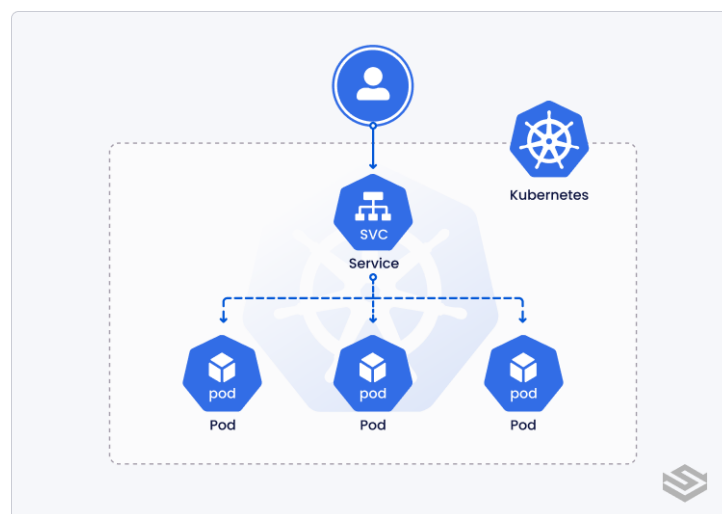
- Deployment giúp quản lý và triển khai các pod. Nó đảm bảo rằng một số lượng pod nhất định luôn được chạy và sẵn sàng tiếp nhận lưu lượng truy cập.

- Deployment cũng hỗ trợ việc cập nhật ứng dụng một cách mượt mà, đảm bảo rằng các phiên bản mới không gây gián đoạn dịch vụ.



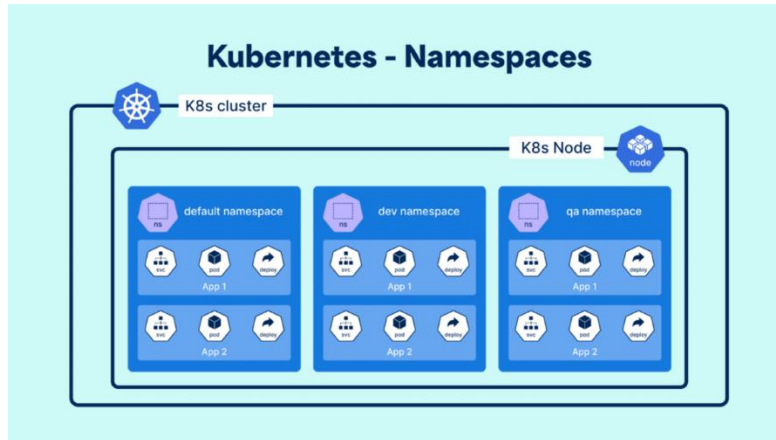
3. Service

- Service là một đối tượng trong Kubernetes để định nghĩa cách các pod giao tiếp với nhau. Service giúp bạn truy cập vào các pod mà không cần phải biết địa chỉ IP cụ thể của chúng.
- Kubernetes cung cấp nhiều loại Service như ClusterIP (mặc định, cho phép truy cập trong nội bộ cluster), NodePort (truy cập từ bên ngoài cluster) và LoadBalancer (sử dụng Load Balancer từ cloud provider).



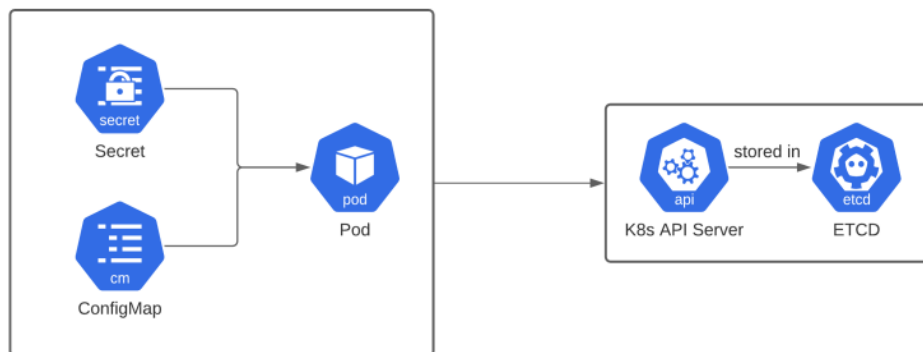
4. Namespace

- Namespace là một cách để phân tách các tài nguyên trong Kubernetes thành các không gian khác nhau. Điều này rất hữu ích khi bạn có nhiều môi trường (như development, staging, production) hoặc nhiều nhóm làm việc khác nhau trong cùng một cluster.



5. ConfigMap và Secret

- **ConfigMap** cho phép bạn lưu trữ các cấu hình dưới dạng các key-value pairs mà không cần phải xây dựng lại image khi thay đổi cấu hình.
- **Secret** là đối tượng bảo mật giúp lưu trữ các thông tin nhạy cảm như mật khẩu, token hoặc chứng chỉ SSL. Secret được mã hóa và có thể sử dụng trong pod mà không cần phải đưa các thông tin nhạy cảm vào trong mã nguồn.



4.3. Các cấu hình cần thiết để triển khai ứng dụng trong Kubernetes cho production

Để triển khai một ứng dụng trong Kubernetes, chúng ta sẽ có rất nhiều cách để triển khai với K8s tuy nhiên sẽ có những cấu hình cơ bản cần có để có thể triển khai được một ứng dụng. Trong đó, một số cấu hình quan trọng cần thiết là:

- **Lập kế hoạch về tài nguyên (Resource Requests & Limits):** Để đảm bảo các container không chiếm dụng quá nhiều tài nguyên, bạn cần định nghĩa yêu cầu tài nguyên cho CPU và RAM cho mỗi Pod. Điều này giúp Kubernetes phân bổ tài nguyên hợp lý và tránh tình trạng cạn kiệt tài nguyên.
- **Quản lý phiên bản ứng dụng (Rolling Updates):** Sử dụng các tính năng như rolling updates trong Deployment để cập nhật ứng dụng mà không làm gián đoạn dịch vụ.
- **Health Checks:** Đảm bảo rằng ứng dụng của bạn có thể được kiểm tra tình trạng hoạt động qua các lệnh livenessProbe và readinessProbe. Điều này giúp Kubernetes có thể tự động tái khởi động ứng dụng nếu phát hiện lỗi.
- **Mạng và bảo mật:** Cần cấu hình các chính sách mạng (Network Policies) và xác thực (RBAC) để đảm bảo rằng các dịch vụ có thể giao tiếp với nhau một cách an toàn và tuân thủ các quy tắc bảo mật.

Dưới đây là ví dụ về cách triển khai một ứng dụng đơn giản:

- **Sử dụng Deployment:** Để triển khai ứng dụng trong Kubernetes, nên sử dụng Deployment thay vì trực tiếp chạy Pods. Deployment giúp dễ dàng kiểm soát việc mở rộng số lượng bản sao của ứng dụng và cập nhật phiên bản mới mà không làm gián đoạn dịch vụ.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-app:v1
          ports:
            - containerPort: 8080

```

- **Cấu hình Service:** Để các ứng dụng trong Kubernetes có thể giao tiếp với nhau, bạn cần cấu hình Service. Ví dụ, nếu bạn muốn expose ứng dụng của mình ra ngoài internet, bạn có thể sử dụng LoadBalancer type Service.

```

apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  type: LoadBalancer
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080

```

- **Sử dụng Secrets và ConfigMaps:** Để bảo mật thông tin nhạy cảm như mật khẩu hoặc API key, nên sử dụng Secrets thay vì hard-code trong mã nguồn. ConfigMaps có thể dùng để cấu hình các giá trị không nhạy cảm như URL hoặc port.

```

apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: YWRtaW4= # (Base64 encoded)
  password: cGFzc3dvcmQ=

```

- **Auto-scaling:** Kubernetes hỗ trợ Horizontal Pod Autoscaler (HPA), cho phép tự động mở rộng số lượng Pods dựa trên các chỉ số như CPU hoặc bộ nhớ. Điều này giúp ứng dụng có thể đáp ứng với lưu lượng truy cập thay đổi.

```

type: ContainerResource
containerResource:
  name: cpu
  container: application
  target:
    type: Utilization
    averageUtilization: 60

```

4.4. Một số lưu ý về bảo mật và quản lý tài nguyên khi triển khai trên Kubernetes

Kubernetes là một công cụ mạnh mẽ, nhưng nếu không được cấu hình đúng cách, nó có thể trở thành mục tiêu cho các cuộc tấn công. Một số lưu ý khi triển khai trên Kubernetes:

- Bảo mật mạng:
 - Sử dụng Network Policies để kiểm soát luồng mạng giữa các Pods. Điều này giúp hạn chế quyền truy cập không mong muốn giữa các ứng dụng trong cluster.
 - Cài đặt Role-Based Access Control (RBAC) để kiểm soát quyền truy cập vào các tài nguyên của Kubernetes.
- Quản lý tài nguyên:
 - Cấu hình Resource Requests và Limits cho mỗi container để Kubernetes có thể phân bổ tài nguyên một cách hợp lý và tránh trường hợp các ứng dụng tiêu thụ quá nhiều tài nguyên, ảnh hưởng đến các ứng dụng khác trong cluster.
 - **Horizontal Pod Autoscaling (HPA):** Kubernetes hỗ trợ tự động mở rộng hoặc thu nhỏ số lượng pod dựa trên tải hệ thống (CPU, memory). Điều này giúp ứng dụng luôn sẵn sàng và tiết kiệm tài nguyên.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2
  template:
    spec:
      containers:
      - name: my-app
        image: my-app:v1
        resources:
          requests:
            memory: "64Mi"
            cpu: "250m"
          limits:
            memory: "128Mi"
            cpu: "500m"
```

- Cập nhật và Rollback:
 - Sử dụng **rolling updates** để triển khai phiên bản mới mà không làm gián đoạn dịch vụ.
 - Nếu có vấn đề trong quá trình cập nhật, Kubernetes cho phép bạn rollback lại phiên bản cũ.
- Mã hóa thông tin nhạy cảm: Sử dụng Secret để lưu trữ các thông tin nhạy cảm và đảm bảo chúng được mã hóa khi lưu trữ.

- **Giám sát và Logging:** Cần triển khai hệ thống giám sát và logging để theo dõi tình trạng hoạt động của các Pod, phát hiện lỗi kịp thời và cải thiện hiệu suất.

5. GitOps và ArgoCD

Trong quá trình triển khai và quản lý ứng dụng, các công cụ tự động hoá như CI/CD đã giúp giảm bớt sự phức tạp và tăng cường hiệu quả. Một trong những xu hướng hiện nay là GitOps, một phương pháp quản lý cơ sở hạ tầng và ứng dụng thông qua Git. Và để thực hiện GitOps trên Kubernetes, ArgoCD là một công cụ cực kỳ hữu ích.

5.1. Định nghĩa GitOps và cách thức hoạt động

GitOps là một phương pháp quản lý hạ tầng và triển khai ứng dụng bằng cách sử dụng Git làm nguồn trung gian duy nhất cho việc quản lý cấu hình hệ thống. Tất cả các cấu hình về hạ tầng và ứng dụng (như các file YAML, Helm charts, hoặc các tài nguyên Kubernetes) được lưu trữ trong các repository Git. Các công cụ tự động như ArgoCD hoặc Flux sẽ theo dõi những thay đổi trong Git và tự động cập nhật hệ thống Kubernetes tương ứng. Nói ngắn gọn thì **GitOps là phương pháp đồng bộ trạng thái hạ tầng hiện tại cho giống với cấu hình hạ tầng được định nghĩa trong các tệp tin IaC lưu trữ trên Git.**



Các luồng công việc của GitOps là:

- Vận hành Git (như cấu hình, cài đặt, cập nhật phiên bản mới của Git, ...)
- Sử dụng Git như nguồn cấu hình chính. Ví dụ: Iac để định nghĩa hạ tầng
- Các hoạt động tạo hoặc thay đổi đều được thực hiện thông qua Git
- Sử dụng CI/CD để đồng bộ hạ tầng từ Git
- Có lưu lại tất cả các thay đổi đã được thực hiện và có thể quay lại thay đổi trước đó

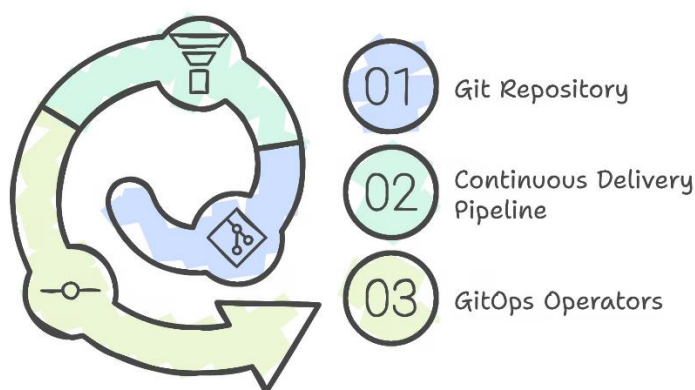
Hiện nay các ứng dụng được phát triển và cập nhật với tốc độ chóng mặt, GitOps ra đời để phần nào đáp ứng cho nhu cầu này.

“GitOps là một khái niệm bổ sung cho DevOps, nó hoạt động như một phần mở rộng của DevOps. GitOps là cách làm việc để nâng cao hiệu quả của DevOps.”

Cách hoạt động của GitOps đa số đều thông qua Git. Ví dụ, khi ta làm việc trong một nhóm DevOps lớn, bây giờ ta cần thay đổi cấu hình của hạ tầng. Ta không có thực hiện thay đổi trực tiếp lên hạ tầng hiện tại mà ta sẽ làm theo quy trình sau:

- Đầu tiên, chúng ta sẽ kéo mã từ Git về máy cá nhân và thực hiện các thay đổi cần thiết trong các tệp tin IaC liên quan đến hạ tầng. Sau khi hoàn tất việc chỉnh sửa, chúng ta sẽ đẩy mã lên Git và tạo yêu cầu merge vào nhánh chính. Yêu cầu này sẽ được xem xét và phê duyệt bởi các thành viên trong nhóm.
- Khi yêu cầu được chấp thuận, Git sẽ thông qua hệ thống CI/CD để tự động cập nhật các thay đổi vào hạ tầng, đảm bảo rằng hạ tầng luôn đồng nhất với các cấu hình trong tệp tin IaC.
- Nếu xảy ra bất kỳ lỗi nào trong quá trình này, chúng ta có thể dễ dàng quay lại trạng thái trước đó bằng cách sử dụng chức năng *Revert* trên Git.

GitOps Workflow Sequence



Ví dụ về cách GitOps hoạt động:

1. Chúng ta cập nhật cấu hình Kubernetes (Deployment, Service, ConfigMap, v.v.) trong Git repository.
2. ArgoCD hoặc Flux sẽ liên tục giám sát repository Git này.
3. Khi có thay đổi, ArgoCD sẽ tự động kéo cập nhật từ Git và áp dụng các thay đổi đó lên Kubernetes cluster.
4. Nếu có sự thay đổi không mong muốn trong Kubernetes cluster, ArgoCD sẽ tự động phục hồi về trạng thái đúng như trong Git repository.

GitOps mang lại nhiều lợi ích, đặc biệt là khi triển khai và quản lý các ứng dụng trên môi trường Kubernetes. Dưới đây là một số lợi ích nổi bật của GitOps:

1. **Tăng tính minh bạch và dễ dàng kiểm soát:**

- Tất cả các thay đổi được ghi lại trong Git, giúp bạn dễ dàng theo dõi lịch sử thay đổi, kiểm tra các phiên bản cấu hình, và duy trì tính toàn vẹn của hệ thống.
- Git cung cấp tính năng kiểm soát phiên bản, giúp bạn quay lại bất kỳ phiên bản nào của ứng dụng hoặc cấu hình nếu gặp sự cố.

2. Cải thiện sự nhất quán:

- GitOps giúp đảm bảo rằng các môi trường khác nhau (development, staging, production) luôn đồng nhất. Các thay đổi cấu hình sẽ được triển khai theo cách tự động và đồng bộ giữa các môi trường này.

3. Tự động hóa triển khai và giảm thiểu lỗi thủ công:

- GitOps tự động hoá quy trình triển khai, giúp giảm thiểu các lỗi do thao tác thủ công. Khi bạn thực hiện thay đổi trong Git, hệ thống sẽ tự động áp dụng các thay đổi đó mà không cần phải can thiệp vào môi trường Kubernetes.

4. Khôi phục tự động và dễ dàng:

- Nếu hệ thống gặp sự cố hoặc có sai lệch với trạng thái trong Git, GitOps có thể tự động phục hồi lại trạng thái đúng của ứng dụng mà không cần phải can thiệp thủ công.

5. Tăng cường bảo mật:

- Việc lưu trữ tất cả cấu hình trong Git cũng giúp bạn dễ dàng kiểm soát quyền truy cập và bảo mật. Các thay đổi được kiểm tra và phê duyệt trước khi được áp dụng.

5.2. ArgoCD: Công cụ GitOps cho Kubernetes

Argo CD là một công cụ mã nguồn mở được thiết kế để hỗ trợ quy trình **GitOps** trong việc triển khai ứng dụng trên **Kubernetes**. Argo CD được áp dụng vào trong bước CD (Continuous Delivery) của luồng CI/CD. Đây là bước ta thực thi triển khai ứng dụng lên trên môi trường máy chủ. ArgoCD giúp tự động hoá việc triển khai và quản lý các ứng dụng Kubernetes thông qua việc giám sát Git repositories. Khi có sự thay đổi trong Git repository, ArgoCD sẽ tự động cập nhật các tài nguyên Kubernetes theo cấu hình mới.

ArgoCD có các tính năng nổi bật như:

- **Quản lý nhiều ứng dụng:** ArgoCD hỗ trợ quản lý và triển khai nhiều ứng dụng từ nhiều repository Git khác nhau.
- **Web UI và CLI:** ArgoCD cung cấp cả giao diện người dùng web (Web UI) và giao diện dòng lệnh (CLI) để bạn có thể dễ dàng theo dõi và quản lý các ứng dụng.

- **Hỗ trợ nhiều loại cấu hình:** ArgoCD không chỉ hỗ trợ Kubernetes manifest (YAML files), mà còn có thể làm việc với Helm charts, Kustomize và các công cụ khác.
- **Tự động đồng bộ và khôi phục:** ArgoCD sẽ giám sát Git repository và tự động cập nhật Kubernetes cluster khi có sự thay đổi. Nếu có sự sai lệch giữa trạng thái thực tế và trạng thái trong Git, ArgoCD sẽ tự động khôi phục lại trạng thái đúng như trong Git.

5.3. Cách thiết lập ArgoCD để quản lý triển khai Kubernetes, quy trình triển khai ứng dụng và tự động hoá cập nhật

Để thiết lập ArgoCD và sử dụng nó trong môi trường Kubernetes, bạn cần thực hiện các bước cơ bản sau:

1. **Cài đặt ArgoCD:** Đầu tiên, cần cài đặt ArgoCD vào cluster Kubernetes. Bạn có thể làm điều này bằng cách sử dụng kubectl để áp dụng các tệp YAML từ kho lưu trữ chính thức của ArgoCD.



```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

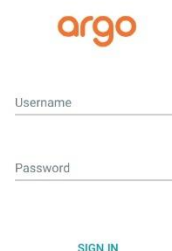
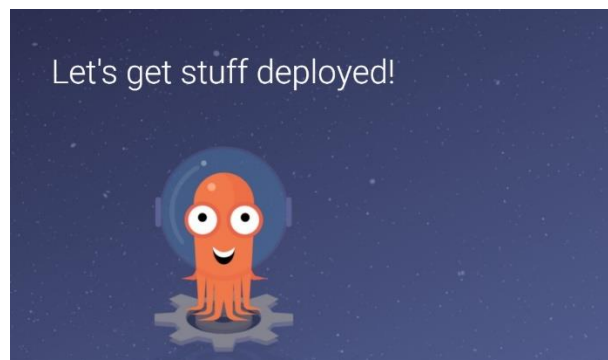
2. Truy cập giao diện web của ArgoCD:

- Sau khi cài đặt, bạn có thể truy cập giao diện web của ArgoCD để dễ dàng quản lý các ứng dụng Kubernetes. Để truy cập giao diện web, bạn cần mở port cho dịch vụ ArgoCD.



```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

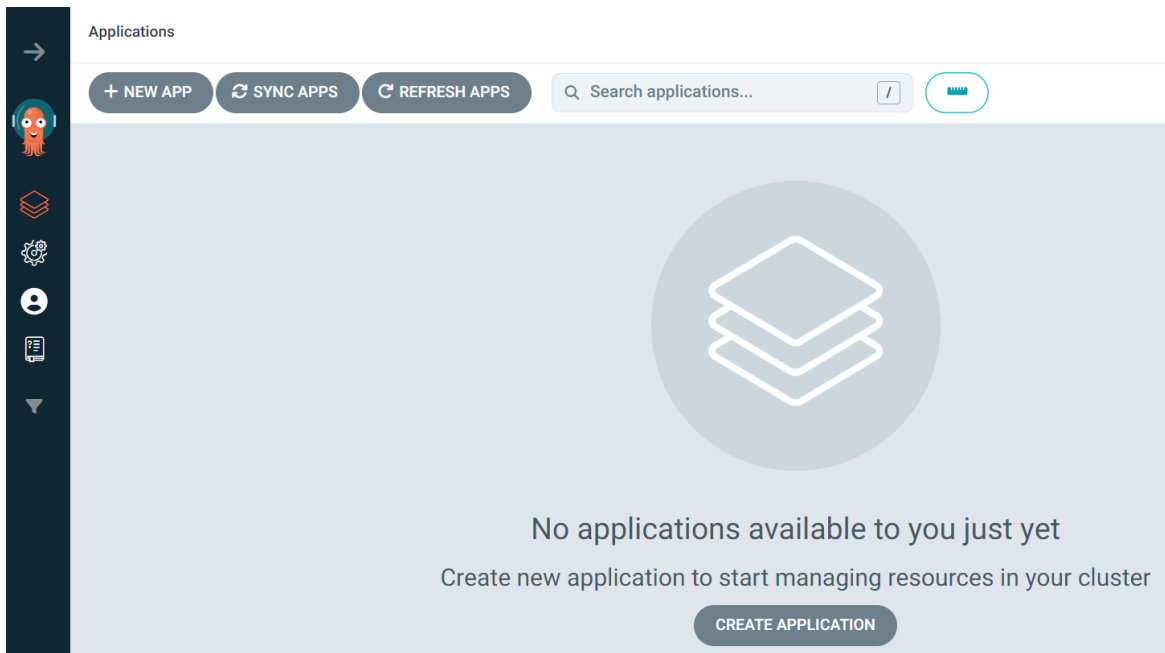
- UI đăng nhập:



- Tài khoản mặc định của Argo CD là admin, để lấy password chạy câu lệnh sau:

```
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d
```

- UI sau khi đăng nhập vào Argo CD:



3. **Kết nối Git repository với ArgoCD:** ArgoCD cần được cấu hình để theo dõi một Git repository nơi chứa các file YAML hoặc Helm charts của ứng dụng Kubernetes.
4. **Tạo ứng dụng trong ArgoCD:** Sau khi Git repository được kết nối, bạn có thể tạo một ứng dụng trong ArgoCD. Cấu hình ứng dụng này sẽ chỉ định nguồn tài nguyên (Git repository) và namespace Kubernetes nơi ứng dụng sẽ được triển khai.

Quy trình triển khai qua ArgoCD và tự động hoá cập nhật thường bao gồm các bước như sau:

1. Cập nhật cấu hình trong Git: Khi cần thay đổi ứng dụng hoặc môi trường, bạn chỉ cần cập nhật các file cấu hình trong Git repository (ví dụ: thay đổi Docker image version, cập nhật tài nguyên Kubernetes).
2. ArgoCD phát hiện thay đổi: ArgoCD sẽ tự động theo dõi Git repository và phát hiện bất kỳ thay đổi nào. Khi có sự thay đổi, ArgoCD sẽ so sánh trạng thái thực tế của hệ thống với trạng thái trong Git repository.

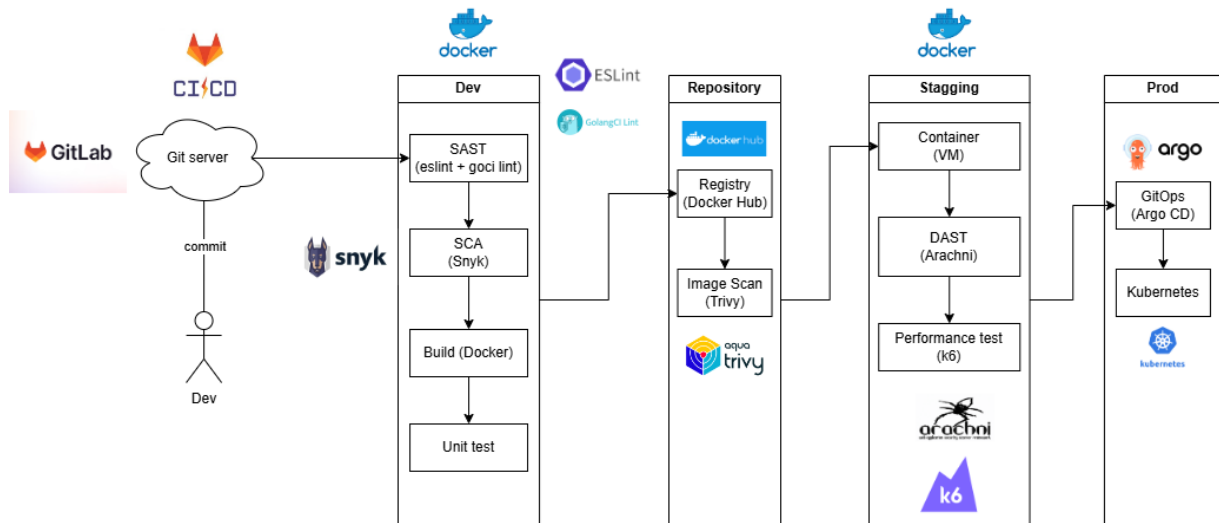
3. Triển khai tự động: Sau khi phát hiện thay đổi, ArgoCD sẽ tự động triển khai ứng dụng lên Kubernetes cluster theo các cấu hình đã cập nhật trong Git. Điều này bao gồm việc tạo hoặc cập nhật các tài nguyên như Pods, Deployments, Services, vv.
4. Theo dõi trạng thái: ArgoCD cung cấp giao diện web để theo dõi trạng thái của các ứng dụng. Bạn có thể dễ dàng xem trạng thái triển khai của các ứng dụng, các lỗi nếu có, và thực hiện rollback nếu cần thiết.

IV. Xây dựng quy trình CI/CD Pipeline toàn diện DevSecOps

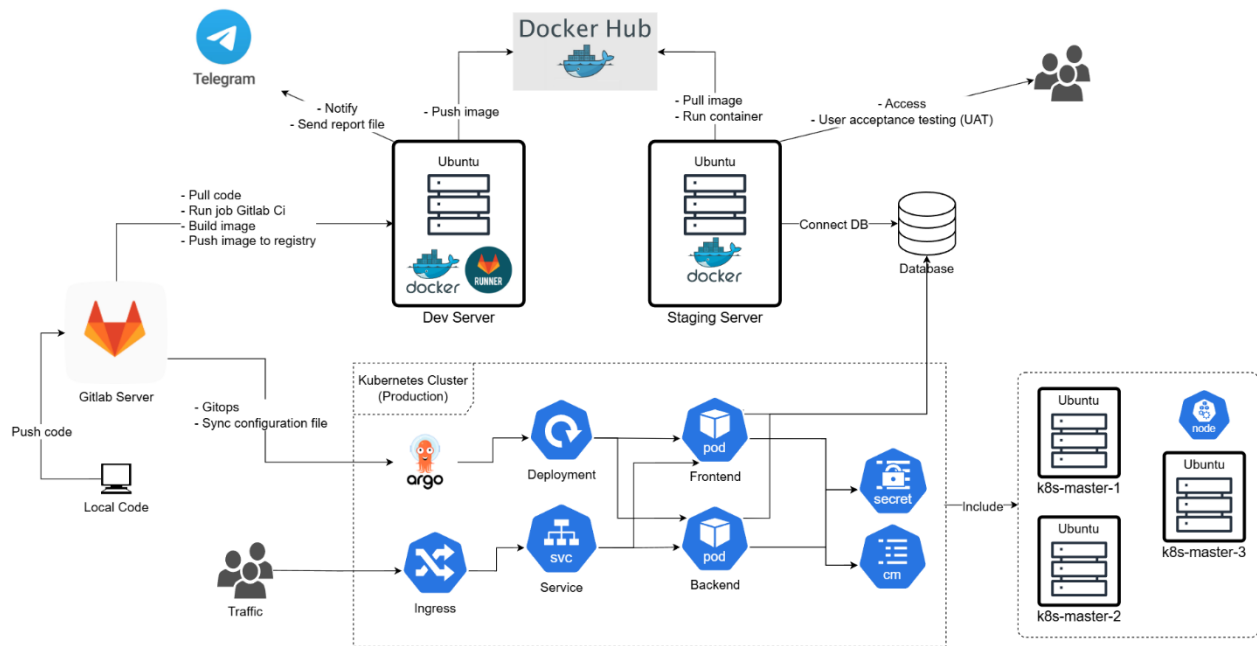
1. Thiết kế quy trình CI/CD DevSecOps

- Mô tả luồng tổng thể:
 - Bắt đầu từ khi developer commit mã lên GitLab Server, đến các bước kiểm thử bảo mật và build image, triển khai lên các môi trường Staging và Production.
- Từng giai đoạn trong Pipeline:
 - Source Code Test: Thực hiện kiểm thử code với các công cụ linting và kiểm thử tĩnh như ESLint, Golang Lint để đảm bảo chất lượng code.
 - Build: Tạo Docker image cho ứng dụng với tên image dựa trên phiên bản commit.
 - Security Scan:
 - SAST & SCA: Kiểm tra bảo mật mã nguồn (với Snyk) và thành phần phần mềm.
 - Image Scan: Sử dụng Trivy để phát hiện các lỗ hổng bảo mật trong Docker image.
 - DAST: Thực hiện kiểm thử bảo mật động với Arachni trên môi trường Staging.
 - Push: Đẩy Docker Image đã build ở giai đoạn build lên Docker hub với tag đã được tạo
 - Performance Testing: Kiểm thử hiệu năng sử dụng công cụ k6 để đảm bảo ứng dụng đáp ứng yêu cầu tải trọng.
 - Deploy: Triển khai ứng dụng lên môi trường staging bằng cách pull docker image từ Docker Hub và chạy bằng Docker container trong 1 máy ảo
 - Send Report: Gửi báo cáo pipeline lên Telegram để thông báo trạng thái.
 - Triển khai lên môi trường Production (Kubernetes) thông qua ArgoCD, tạo merge request mới với việc thay đổi tag của image trong deployment bằng tag của image đã được build trong giai đoạn “Build”
 - Kiểm tra trạng thái sync ArgoCD và xem ứng dụng đã được deploy trên Production với version mới nhất chưa
 - Tiếp tục theo dõi ứng dụng trên Production thông qua Kubernetes Dashboard.

Sơ đồ quy trình CI/CD pipeline Devsecops:



Sơ đồ cơ sở hạ tầng của quy trình:



2. Thiết Lập Môi Trường Phát Triển và CI/CD

- **GitLab Server & GitLab CI:**
 - Cấu hình GitLab Runner cho các job trong pipeline.
 - Cấu hình các biến môi trường như `USER_PROJECT`, `PATH_PROJECT`, và các token bảo mật cần thiết cho pipeline.
- **Docker & Docker Hub:**
 - trên Dev Server và cấu hình để đẩy image lên Docker Hub.
- **Kubernetes Cluster (Production):**
 - Cấu hình cluster Kubernetes với các node.

- Cấu hình Argo CD cho phép triển khai GitOps, đảm bảo tự động đồng bộ ứng dụng.
- Staging Server:
 - Thiết lập máy ảo cho container chạy thử nghiệm trước khi triển khai Production.
- Telegram Notification:
 - Cấu hình bot Telegram để nhận thông báo và báo cáo từ pipeline.

3. Các Bước Xây Dựng Quy trình CI/CD Thực tế

- Giai đoạn Test Source Code:
 - Linting (Job lint): Cấu hình job lint với Node.js và ESLint, lưu trữ báo cáo.
 - SCA với Snyk: Sử dụng Docker để chạy Snyk scan và lưu trữ báo cáo.
- Giai đoạn Build:
 - Cấu hình job build Docker image với tên phiên bản dựa trên commit.
- Giai đoạn Security Scan Image:
 - Trivy: Thiết lập job để scan Docker image với Trivy, lưu báo cáo dưới dạng HTML.
- Giai đoạn Push:
 - Push Docker image với tên phiên bản dựa trên commit ở giai đoạn “Build” lên Docker hub.
- Giai đoạn Deploy:
 - Tạo và chạy container mới với image đã build lên môi trường Production.
- Giai đoạn DAST:
 - Arachni: Sử dụng Docker container của Arachni để scan trang web trên Staging.
- Giai đoạn Performance Testing:
 - k6: Chạy script kiểm thử hiệu năng bằng k6 để đánh giá khả năng chịu tải của ứng dụng.
- Giai đoạn Send Report:
 - Gửi báo cáo kiểm thử từ các giai đoạn trên lên Telegram.
- Giai đoạn tạo Merge Request cho Repository chứa cấu hình Kubernetes cho ứng dụng
- Giai đoạn ArgoCD tiến hành sync với main của Repository chứa cấu hình Kubernetes và kiểm tra xem ứng dụng được deploy với phiên bản mới nhất.
- Giai đoạn theo dõi ứng dụng trên Kubernetes

V. Kết luận và hướng phát triển

1. Tóm tắt quy trình CI/CD và DevSecOps

Quy trình **CI/CD (Continuous Integration / Continuous Delivery)** là một phương pháp giúp tự động hoá việc phát triển phần mềm từ việc kiểm tra mã nguồn, tích hợp đến việc triển khai ứng dụng lên môi trường sản xuất. CI tập trung vào việc tích hợp mã nguồn liên tục từ các lập trình viên vào kho mã nguồn chính (repository), trong khi CD tự động hoá việc đưa phần mềm từ môi trường phát triển lên môi trường thử nghiệm hoặc sản xuất.

DevSecOps là một mô hình phát triển phần mềm hiện đại kết hợp **DevOps** và **Security**. DevSecOps giúp tích hợp bảo mật vào quy trình phát triển ngay từ đầu, đảm bảo rằng bảo mật không phải là công việc riêng biệt mà là một phần của quy trình tự động. Với việc triển khai DevSecOps, chúng ta có thể giảm thiểu các rủi ro bảo mật và tăng cường tính ổn định của phần mềm.

Trong đồ án này, đã triển khai quy trình DevSecOps với các công cụ như **Docker**, **GitLab CI**, và **Kubernetes**, mỗi công cụ đóng một vai trò quan trọng trong việc xây dựng và duy trì quy trình CI/CD.

- **Docker** giúp đóng gói ứng dụng vào các container, đảm bảo tính nhất quán và dễ dàng triển khai trên các môi trường khác nhau.
- **GitLab CI** cung cấp nền tảng tự động hoá kiểm tra mã nguồn, xây dựng và triển khai ứng dụng thông qua các pipeline.
- **Kubernetes** hỗ trợ việc triển khai, mở rộng và quản lý các container trên môi trường đám mây, giúp đảm bảo tính sẵn sàng và khả năng mở rộng của ứng dụng.

Điểm mạnh của quy trình này là giúp giảm thiểu lỗi do con người, tối ưu hoá thời gian triển khai, đồng thời cải thiện khả năng bảo mật nhờ việc tích hợp các bước kiểm tra bảo mật vào trong pipeline.

2. Những thách thức và giải pháp khi triển khai DevSecOps trong dự án

Mặc dù việc triển khai DevSecOps với Docker, GitLab CI và Kubernetes mang lại nhiều lợi ích, nhưng trong quá trình thực hiện vẫn gặp phải một số **thách thức**:

- **Quản lý và cấu hình các container Docker:** Việc quản lý hàng nghìn container trên môi trường sản xuất là một vấn đề phức tạp. Để giải quyết, chúng ta có thể sử dụng các công cụ hỗ trợ như **Docker Compose** để quản lý các dịch vụ và ứng dụng container hoá dễ dàng hơn.
- **Xử lý các lỗi bảo mật trong quá trình CI/CD:** Quá trình tự động hoá sẽ dễ dàng bỏ qua các lỗi bảo mật nếu không được cấu hình đúng. Một giải pháp là tích hợp

các công cụ kiểm tra bảo mật tự động như **Snyk** hay **Clair** vào trong pipeline CI để phát hiện các lỗ hổng bảo mật từ giai đoạn đầu.

- **Khả năng mở rộng với Kubernetes:** Mặc dù Kubernetes rất mạnh mẽ trong việc quản lý container, nhưng việc cấu hình và tối ưu hoá các cluster Kubernetes để phù hợp với môi trường sản xuất vẫn có thể gặp khó khăn, đặc biệt là khi triển khai các ứng dụng quy mô lớn. Việc sử dụng các công cụ hỗ trợ như **Helm** để quản lý các biểu đồ Kubernetes có thể giúp giảm bớt sự phức tạp trong việc triển khai.

Các **giải pháp** cho các thách thức này bao gồm:

- **Sử dụng các công cụ giám sát và logging:** Các công cụ như **Prometheus**, **Grafana**, và **ELK stack (Elasticsearch, Logstash, Kibana)** giúp giám sát và thu thập log từ các ứng dụng đang chạy trên Kubernetes, giúp dễ dàng phát hiện sự cố.
- **Tự động kiểm tra mã nguồn:** Cấu hình các job kiểm tra mã nguồn trong GitLab CI giúp phát hiện lỗi ngay từ giai đoạn đầu, giảm thiểu việc lỗi phát sinh trong quá trình triển khai.
- **Đảm bảo tính bảo mật thông qua kiểm tra lỗ hổng bảo mật:** Sử dụng các công cụ như **Trivy** hay **Aqua Security** để quét các container trong pipeline CI, đảm bảo rằng không có lỗ hổng bảo mật được đưa vào môi trường sản xuất.

3. Hướng phát triển trong tương lai

Mặc dù quy trình CI/CD và DevSecOps đã giúp tự động hoá và cải thiện hiệu quả phát triển phần mềm, nhưng công nghệ và yêu cầu phát triển phần mềm luôn thay đổi. Các **xu hướng và công nghệ mới** có thể giúp cải thiện quy trình DevSecOps trong tương lai bao gồm:

- **AI và Machine Learning trong CI/CD:** Việc áp dụng **AI** vào các pipeline CI/CD sẽ giúp phát hiện lỗi sớm và tối ưu hóa các quyết định triển khai. Ví dụ, AI có thể dự đoán các lỗi tiềm ẩn trong mã nguồn hoặc trong quá trình triển khai, từ đó tự động đưa ra các cảnh báo hoặc thậm chí tự động sửa lỗi.
- **Serverless Computing:** Thay vì phải quản lý và duy trì các máy chủ truyền thống, việc chuyển sang mô hình **serverless** có thể giúp giảm bớt gánh nặng về cơ sở hạ tầng và tăng cường tính linh hoạt cho quy trình CI/CD.
- **Infrastructure as Code (IaC):** Việc sử dụng **IaC** với các công cụ như **Terraform** hoặc **Ansible** giúp tự động hoá việc cấu hình hạ tầng, từ đó hỗ trợ dễ dàng hơn trong việc mở rộng quy trình DevSecOps.

Cải tiến và mở rộng quy trình DevSecOps cũng có thể bao gồm việc tích hợp thêm các công cụ bảo mật mạnh mẽ hơn, như **Security as Code** (Security policies as code), giúp tự động hoá các chính sách bảo mật và đảm bảo rằng chúng luôn được áp dụng đúng đắn trong suốt vòng đời phát triển phần mềm.

Để đáp ứng các yêu cầu ngày càng cao của các ứng dụng quy mô lớn, quy trình DevSecOps có thể được mở rộng để hỗ trợ **multi-cloud** và **hybrid-cloud environments**, giúp triển khai ứng dụng trên nhiều nền tảng đám mây khác nhau mà không gặp phải các vấn đề tương thích.

TÀI LIỆU THAM KHẢO

1	"The DevOps Handbook" – Gene Kim, Patrick Debois, John Willis, Jez Humble.
2	"DevSecOps: A leader's guide to producing secure software without compromising flow, feedback and continuous improvement" – Jim Bird
3	DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations – Gene Kim, Patrick Debois, John Willis, Jez Humble.
4	Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation – Jez Humble, David Farley
5	https://viblo.asia/p/tim-hieu-devops-va-mot-so-kien-thuc-co-ban-khi-moi-bat-dau-Qbq5Q6L3KD8
6	"Continuous Integration and Continuous Deployment" - https://www.atlassian.com/continuous-delivery
7	"OWASP: SAST - Static Application Security Testing" - https://owasp.org/www-project-sast/
8	"Software Composition Analysis: What is it and Why it's Important" - https://www.synopsys.com/software-integrity/software-composition-analysis.html