universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

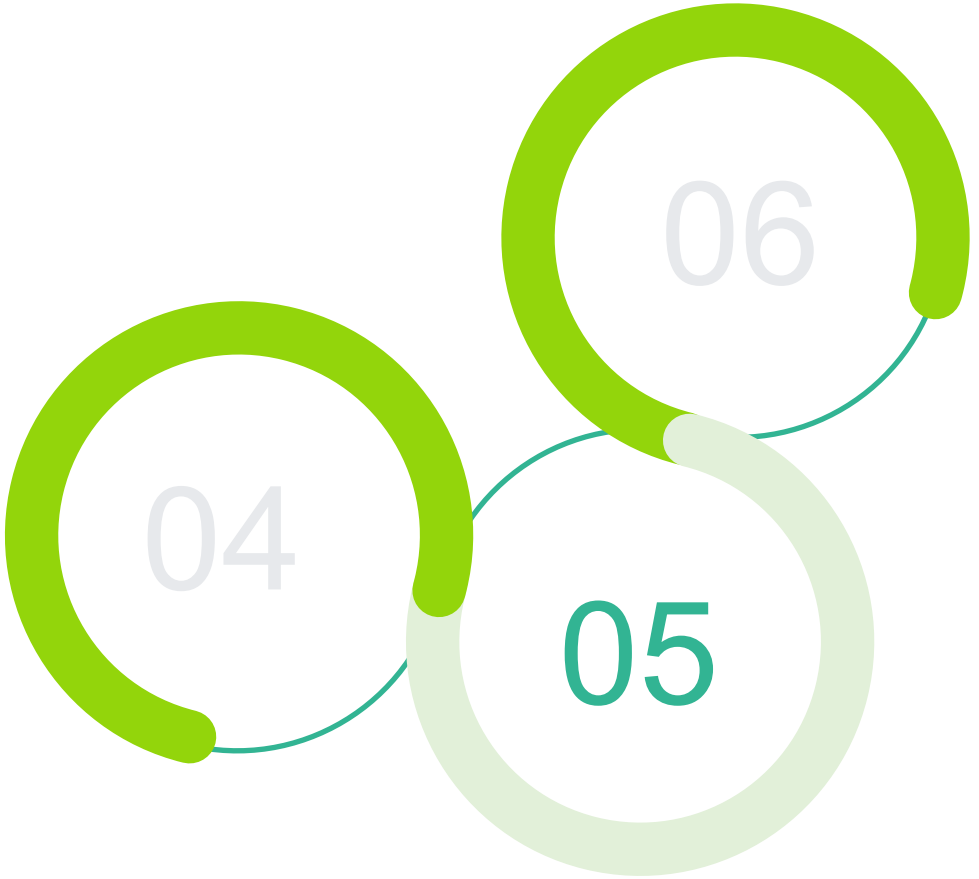
Software Engineering

Strategies for continuous delivery

2025/2026

Outline

- Core principles of software delivery
- Feature toggling
- Progressive delivery strategies
- Continuously releasing user-installed software



Minimal Viable Product

- Scenario: The client provided ideas, and the team developed the core features.

- Current status:
  - Core functionality is **implemented and working.**
  - Product is **usable by early adopters** for testing and feedback.
  - Scope is **minimal**, focusing only on **essential features**.

Release candidate

- A change may or may not be releasable
  - The code alone doesn't tell us.
- The build, deployment, and test process validates whether a change can be safely released.
  - This process increases confidence that the change is safe for production.
- Applies to any change: new functionality, bug fixes, or performance tuning.

Core principles of software delivery

Create a repeatable, reliable process for releasing software

- The repeatability and reliability derive from two principles:
  - **Automate** almost everything
  - Keep everything you need to build, deploy, test, and release your application in **version control**

- Deploying software ultimately involves three things:
  - Provisioning and managing the environment in which your application will run
    - Hardware configuration, software, infrastructure, and external services
  - Installing the correct version of your application into it
  - Configuring the application, including any data or state it requires

- Application deployment can be implemented using a fully automated process from version control
  - Application configuration can also be a fully automated process, with the necessary scripts and state kept in version control or databases

Core principles of software delivery

Automate almost everything

- Some tasks **cannot be automated**
  - Exploratory testing, user demos, compliance approvals
- Most of the release process **can be automated**
  - Builds, deployments, acceptance tests, database migrations, network/firewall configs
- Automation reduces repeated manual effort and ensures consistency
- **Start gradually**
  - Automate bottlenecks first, expand over time
- Automation is essential for an efficient deployment pipeline
  - People get what they need at the push of a button

Core principles of software delivery

Keep everything in version control

- Keep everything needed to build, deploy, test, and release under version control:
  - Requirement documents, test scripts, automated tests
  - Deployment, database, network, and stack configuration scripts
  - Libraries, toolchains, and technical documentation
- Each build should have a single identifier linking all relevant files
  - e.g., build number or change set
- A new team member should be able to:
  - Check out the repository
  - Run a single command to build and deploy the application

Core principles of software delivery

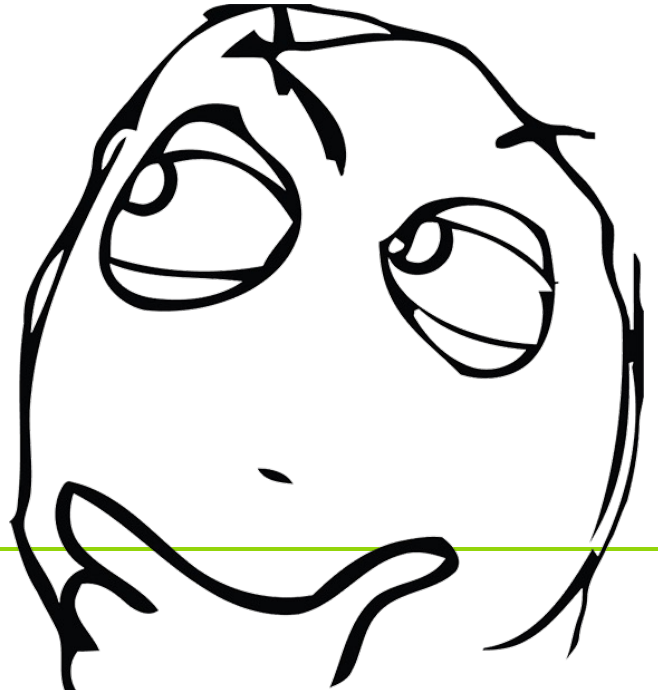
Done means released

- **There is no “80% done.”**
  - Things are either done, or they are not.
- It is possible to estimate the work remaining before something is done
  - But those will only ever be estimates.
- This principle has an interesting corollary
  - It is not in the power of one person to get something done.
  - It requires a number of people on a delivery team to work together to get anything done.
  - That’s why it’s so important for everybody to work together from the beginning.
- It’s also why the whole delivery team is responsible for delivering

Software delivery

- Same scenario: Client liked the product, now he wants more features.
  - How should we deliver new features to him?
  - Should we release everything at once or gradually?
  - Do we force updates, or give users a choice?
  - How do we minimize risk while keeping users happy?
  - What metrics or feedback should guide our release decisions?

- **Would you rather risk upsetting a few users with automatic updates or risk losing everyone by delaying new features?**



Feature toggling

Overview

- Feature toggling (also called feature flags) is a software development technique that allows features to be **turned on or off** without deploying new code.
  - Instead of releasing a new feature to all users immediately, it enables control over who can access it and when.

```
if feature_flag_enabled("new_dashboard"):
    show_new_dashboard()
else:
    show_old_dashboard()
```

Feature toggling

Motivation

- **Continuous delivery**
  - Ship code to production even if features aren't ready.
- **A/B testing & experimentation**
  - Test features on a subset of users.
- **Safe rollbacks**
  - Disable a feature instantly if issues appear.
- **Targeted releases**
  - Roll out features to specific users, regions, or accounts.
- **Decouple deployment from release**
  - Code can be deployed independently of feature activation.

Feature toggling

Four main types

- Release toggles
  - Control **when a feature is visible**.
  - Temporary; removed after full release.
  - Example: Releasing a new payment system to 10% of users.
- Experiment toggles
  - For **A/B testing**.
  - Determine which variant of a feature performs better.
  - Example: Two versions of a signup form.
- Ops toggles (Operational toggles)
  - Control **operational aspects** rather than functionality.
  - Useful for **performance tuning, enabling/disabling features under load**.
  - Example: Toggle caching or logging level.
- Permission toggles (or permanent toggles)
  - Enable features for **specific users or groups**.
  - Example: Premium features enabled for paying customers.
  - Can be **long-lived**.

Feature toggling

Four main types

- Release toggles
  - Control **when a feature is visible**.
  - Temporary; removed after full release.
  - Example: Releasing a new payment system to 10% of users.
- Experiment toggles
  - For **A/B testing**.
  - Determine which variant of a feature performs better.
  - Example: Two versions of a signup form.
- Ops toggles (Operational toggles)
  - Control **operational aspects** rather than functionality.
  - Useful for **performance tuning, enabling/disabling features under load**.
  - Example: Toggle caching or logging level.
- Permission toggles (or permanent toggles)
  - Enable features for **specific users or groups**.
  - Example: Premium features enabled for paying customers.
  - Can be **long-lived**.

Continuously delivery

Progressive delivery strategies

Eat you own dog food

- **Idea**: Dev team (and internal staff) use the newest version internally

- If everything is as expected it is roll it out to further users

Progressive delivery strategies

Canary releases

- **Idea**: release a new version to a subset of users first, while all other users interact with the old, stable version
  - In case of issues with new version, only a small number of users is affected

- The name comes from the “canary in a coal mine”
  - Miners used canaries to detect toxic gases before it was too late. If the canary showed signs of distress, the miners knew there was danger — the same principle applies here.

- User selection based on:
  - (geographic) location
  - role (admin, early, stable, etc.)
  - random basis (e.g., 1% of traffic)

Progressive delivery strategies

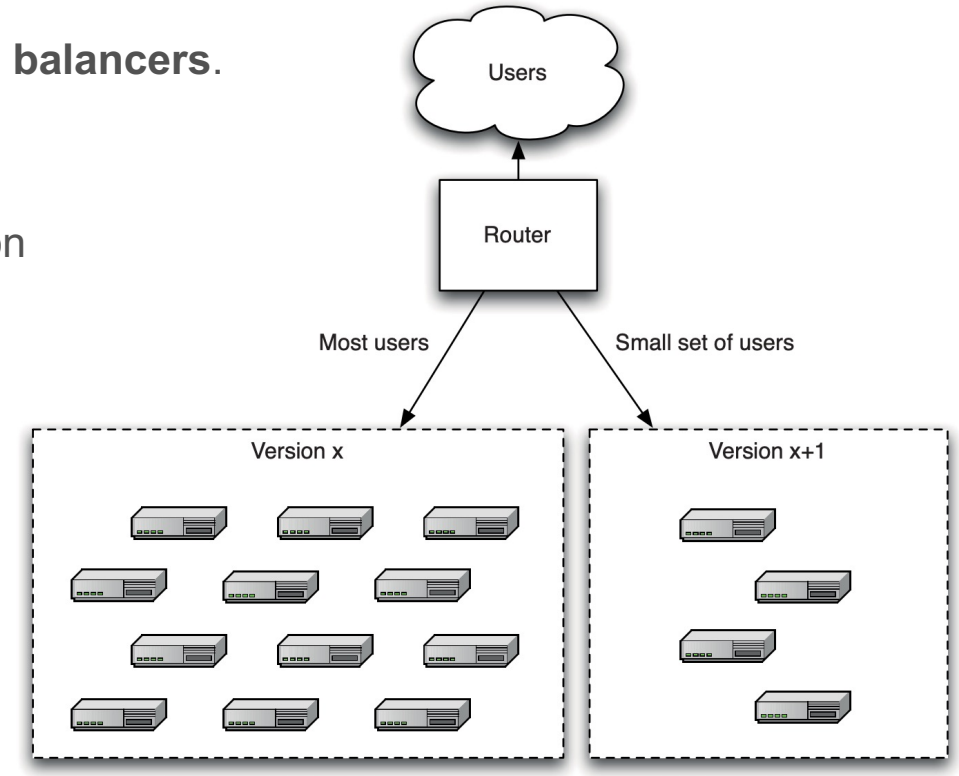
Canary releases

- Canary releases can be implemented at **different levels**:

a) Infrastructure / Load Balancer Level
   - Traffic is routed selectively using **reverse proxies** or **load balancers**.

b) Application level
   - The application decides which users get the canary version
     - (based on user ID, region, feature flag, etc.).
   - Often combined with **feature toggles**.

c) CI/CD Pipeline Level
   - Orchestrated directly in the pipeline.



Progressive delivery strategies

Canary releases

- Monitoring is *crucial* for canary releases. Key indicators include:
  - Error rates
  - Response times
  - CPU/memory usage
  - User satisfaction metrics
  - Business KPIs (e.g., conversion rate, retention)

- Automation can compare canary metrics to the baseline and **automatically decide whether to continue or rollback**
  - Known as Automated Canary Analysis (ACA)

Progressive delivery strategies

Canary releases

- Let's say version 2.0 of an API is ready.
  - Step 1: Deploy it to **5% of users**.
  - Step 2: Monitor **latency**, **error rate**, **CPU load**, etc.
  - Step 3: If metrics are healthy → increase rollout to **20%, 50%, 100%**.
  - Step 4: If metrics degrade → **rollback to version 1.9** for everyone.

- Key Objectives
  - Reduce risk of production failures.
  - Validate functionality under real-world conditions.
  - Gather early feedback from a small audience.
  - Enable fast rollback if the release introduces issues.

Progressive delivery strategies

Dark launches

- A deployment strategy where a feature is released to production
- Feature is **not yet exposed** to end-users
- Used to test scalability, performance, and stability
- Often combined **with feature toggles** or **hidden endpoints**

this allows us to monitor the load on existing systems before the feature is officially released

Progressive delivery strategies

Dark launches

- Identify **performance bottlenecks** before release
- Detect infrastructure or **integration issues**
- Validate **resource usage under real conditions**
- Ensure smooth user experience once the feature is public
- Core idea
  - Deploy new features **into production**
  - Keep them **invisible to users**
  - Observe how they behave under real production load

Progressive delivery strategies

Gradual rollouts

- Gradually increase the number of users assigned to the newest version
- Continue the rollout step by step until it fully replaces the previous version
- Allows early detection of issues before full release

Progressive delivery strategies

Gradual rollouts

- Workflow
  - Start with a small subset of users
  - Monitor performance and errors
  - Gradually expand user percentage
  - Once stable, complete the rollout

- Purpose
  - To test scalability and stability under real-world conditions
  - To observe system behaviour with increasing load
  - To minimise risk of widespread failure

Progressive delivery strategies

A/B Testing

- A/B Testing compares two or more versions of a feature or interface
- Users are randomly divided into groups (A, B, etc.)
- Each group experiences a different version
- Results determine which version performs better

Progressive delivery strategies

A/B Testing

Purpose

- To evaluate user behaviour and measure impact of design or functional changes
- To validate hypotheses before full rollout
- To support evidence-based decision-making

Process Overview

- Define a goal or success metric
  - (e.g., click rate, engagement, conversion)
- Split users into test groups
- Deploy each variant
- Collect and analyse data statistically
- Decide which version to adopt or discard

all users same version

50% users new version
50% users old version

Progressive delivery strategies

A/B Testing

- Can be used within a progressive rollout to **test user reactions**
- A/B Testing focuses **on comparison**, while progressive rollout focuses on scale and stability
- Combining both ensures functional and performance validation
- A/B Testing helps organisations:
  - Make data-backed decisions
  - Understand user preferences
  - Deliver better-performing features

Progressive delivery strategies

Blue/Green Deployments

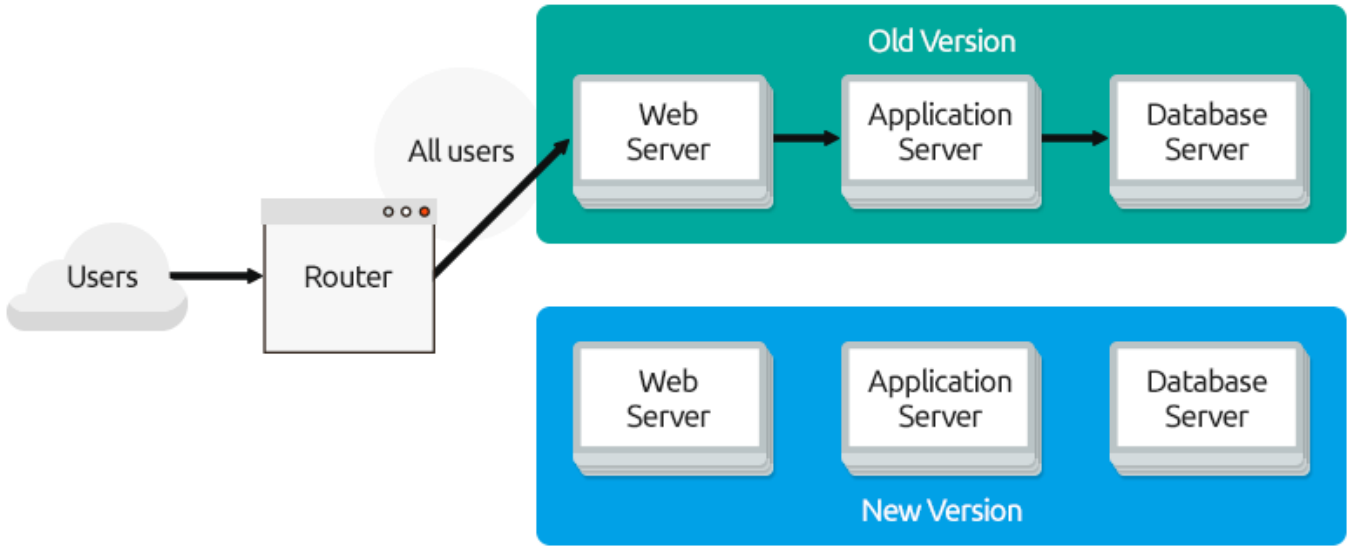
- Maintain two identical production environments:
  - Green: current, stable version
  - Blue: new version under preparation
- The router directs traffic to either environment
- Switching traffic effectively releases the new version

Progressive delivery strategies

Blue/Green Deployments

- Release process

1. Deploy the new version to the inactive environment (e.g., Blue)
2. Test the new version in isolation
3. Once validated, switch traffic from Green → Blue
4. Keep the previous environment ready for instant rollback



Progressive delivery strategies

Blue/Green Deployments

Advantages

- Enables **zero-downtime releases**
- Supports **quick rollback** in case of errors
- Simplifies testing under production-like conditions

Considerations

- Databases are shared or duplicated across environments
- Switching requires data migration or synchronization
- Consistency and latency must be carefully managed

Progressive delivery strategies

Relationship between strategies

Strategy	Key Focus	Rollback Type	User Exposure
Progressive Rollout	Gradual exposure	Partial	Controlled increase
Canary Release	Early validation	Partial	Selected users
Blue-Green Deployment	Full environment swap	Instant	Complete switch
A/B Testing	Behavioural comparison	N/A	Parallel testing groups

Continuously releasing user-installed software

Continuously user software releases

- Releasing a new version of your application to a production environment you control is one thing.
- Releasing a new version of software installed by users on their own machines (client-installed software) is another.
- There are several issues to consider:
  - Managing the upgrade experience
  - Migrating binaries, data, and configuration
  - Testing the upgrade process
  - Getting crash reports from users

Continuously user software releases

Problem

- Multiple versions of software “in the wild” cause support headaches.
- Debugging requires reverting to specific historical versions.
- Ideally: all users run the latest stable version.
- Goal: make upgrades painless and seamless.

Continuously user software releases

Upgrade strategies (client)

- Check for new version, prompt user to download & upgrade.
  - Easy to implement, frustrating for users.
- Download in background, prompt for installation.
  - Less intrusive, but repeated prompts can annoy.
- Download silently, upgrade on restart (Firefox model).
  - Most user-friendly if done correctly.

Continuously user software releases

Pitfalls of user choice

- Users often **delay upgrades** due to fear of breaking software.
- Providing choice can signal lack of developer confidence.
- Rational user decision: avoid upgrading if process seems risky.

Continuously user software releases

Correct approach

- Make upgrades bulletproof and silent.
- Automatically revert to previous version if upgrade fails.
- Report failures to development team without bothering the user.
- Only prompt users if corrective action is needed.

Continuously user software releases

Don’t delete the old files, move them

- Make sure you keep a copy of the previous version around
  - Then, ensure that you clear out the old files before deploying the new version.
  - If a stray file from the old deployment is still lying around in the newly deployed version, it can cause hard-to-track-down bugs.

- A good practice in the UNIX world is to **deploy each version** of the application into a **new directory** and **have a symbolic link** that points to the current version
  - Deploying and rolling back versions is simply a matter of changing the symbolic link to point to the previous version.

- The network version of this is to have different versions sitting on different servers or different port ranges on the same server.
  - Switch between them using a reverse proxy (Blue-Green deployments)

Practical Guide

Practical Guide

- Install a local version to learn and later deploy it on your server.
  - https://github.com/flagsmith/flagsmith
- Quick start
  - https://docs.flagsmith.com/quickstart