Architecture in the Age of Continuous Delivery

Jenish Paudel

January 2026

What is Continuous Delivery?

Continuous Delivery is a software development practice in which software is always in a releasable state. This means:

- The application is deployable at any point in it's life cycle.
- Teams prioritize keeping the system deployable over adding new features.
- Every change goes through automated build, test and integration steps, ensuring it is production ready.

Continuous Deployment extends the idea behind Continuous Delivery by also automating the release to production. Whenever a change passes all automated tests, it is automatically deployed to the live environment. No manual approval step is required.
Important Distinction: Continuous Delivery is always ready for production but manual decision is needed to release while Continuous Deployment is automatically released to production.
Continuous Deployment is practically impossible and even dangerous in many areas. (Ex. Finance, Healthcare, Traffic Control etc.) but it is used by big companies such as Meta, Amazon and Google who deploy hundreds or thousands of changes per day. But these companies also have massive and expensive well-engineered systems that allow this. In reality most rely on Continuous Delivery. This article is written with Continuous Delivery in mind, not deployment and "CD" acronym refers to Continuous Delivery. Not Deployment.

Why Continuous Delivery?

With traditional approach, changes happen every few weeks, or months. When releases are infrequent, releases become massive, failures become harder to debug and deployment days become stressful. CD aims to solve this problem by keeping software, always in a releasable state, with small incremental changes instead of giant risky ones. It also provides rapid automated testing feedback, which help improve the developmental process. Manual tests can take hours or days, while automated tests can run in minutes and remove possibility of human errors.

What Does Building Architecture for Continuous Delivery Imply?

Building Architecture for Continuous Delivery means a greater focus on specific Architecturally Significant Requirements and subsequent Quality Attributes. These ASRs and QAs are not born out of user's requirements but rather from the desire to adhere to the continuous delivery organizational structure. Ultimately, the implementation of continuous delivery will cause extra challenges, concerns and costs when designing architecture to fit into systems that want to implement continuous delivery.

Introduction to Software Architecture in CD

The fundamental limitations to adopting CD practices are deeply ingrained in the architecture of a system. In CD environments, the architecture must support essential attributes when enabling rapid and reliable feedback loops between operations and development. Architectures need to emphasize loose coupling, high cohesion, service autonomy, observability and deployability.

Examining Monoliths CD

A Monolith system is defined as a single, large executable artifact that bundles multiple components into one tightly integrated, deployable unit. All functionalities, business logic and services are packaged and released as a single application. Due to this, every change requires rebuilding, retesting and redeploying the entire system.
Monolithic systems don't make CD impossible but are inherently a barrier to Continuous Delivery. It is extremely difficult, time-consuming and expensive to do so. Developers are forced to wait for long build times and heavy test runs, which heavily constraints the effectiveness of CD.

Monolithic Dependency

As monolithic systems grow in size and complexity, the internal components and layers become tighter. This is a major obstacle as CD relies on rapid, incremental changes that can be tested, validated and deployed with minimal risk. Every deployment requires reanalyzing and retesting the entire dependency structure. The effort to maintain this level of dependency awareness slows down the deployment pipeline, increases operational risk and reduces confidence in frequent releases.

Monolithic Teams

The architecture of a software system tends to mirror the communication and organizational structure of the teams that build it. In an environment where multiple teams contribute to a single monolithic application, cross-team dependencies create friction throughout the CD pipeline. Even if one team has highly automated and reliable code, they can still be blocked and destabilized by the work of the other teams, contributing to the same monolith. When adopting CD, it becomes crucial to address the team-structure challenges that arise from inter-dependencies as coordination overheads must constantly synchronize and agree on release timing. Any lack of communication leads to a failed build and blocked pipeline.

Monolithic Feedback

Continuous Delivery relies heavily on fast and frequent feedbacks after every change a developer makes. This enables teams to detect issues and validate assumptions quickly. However, monolithic architectures inherently slow down this feedback loop. Because modules in a monolith are tightly coupled, changes in one area may trigger or require retesting across many components. As the dependencies expand, the number of test cases also increase dramatically, which results in slower and less-efficient development cycle.

Monolithic Automation

The heavy-weight nature of monolithic systems make it difficult to fully automate builds, testing and processes required for Continuous Delivery. As application grows, the automation scripts also grow larger and more complicated. This makes automation harder to maintain and more prone to failures. If automated pipeline for a monolith is slow and brittle, the organization struggles to achieve fast, reliable and repeatable delivery cycles that CD demands.

Moving beyond Monoliths

In conclusion from above points, Continuous Delivery works best when the system can be built, tested and prepared for release interdependently at the component level. Each part of the system should be able to move through the pipeline without being blocked by changes in other parts. Four important factors that matter with CD when splitting a monolith or designing a system are:

- Deployability: Each component should be able to move through the CD pipeline and become ready for deployment without requiring changes to the rest of the system.
- Testability: Fast, reliable automated tests. Each component should be easy to test in isolation

- Scalability: Pipeline should be small maintainable. No component should require full-system builds Each component should scale on it's own.
- Modifiability: Changes in one component should not affect other components.

Other helpful criterions to consider:

- One business domain per component: A component/service should focus on one business capability. This makes it easy to test, deploy and reason about independently.
- Low Dependency: A service should not depend heavily on another.
- Automation Compatibility: A component should be built small enough such that it can be automatically tested and be set for deployment. Slow components kill CD. Small and simple means fast pipelines, which means fast feedback.
- Team Autonomy: One team should be able to fully own the component. This means, given one component, one team should understand its code, build it, test it, debug it and deploy it. CD breaks if a team has to wait on other team, if coordination slows deployment or if component knowledge is too scattered.
-

Common Architectural Approaches

Two notable architectural approaches that achieve the priorly listed criterions are Vertical Slicing Microservices.

Vertical Slicing:

Instead of structuring the architecture separate layers like UI layer, service layer, data layer, etc. Vertical Slicing bundles these layers together inside feature-specific modules. For example:

- Order module
- User Module
- Post Module, etc.

Each module/slice contains:

- it's own UI components
- its own business logic
- it's own test

- it’s own deployment artifacts

Vertical slicing is often the first step organizations take when modernizing their monolithic applications. By carefully looking at the coupling of the systems, they can be carefully separated into slices.

Micro-Services

Microservices go one step further from Vertical Layering, by splitting the system into fully separate services, each running it’s own process and deployed independently. Microservices are truly independently deployable with small, isolated codebases that run faster builds and tests. Failure are extremely contained and enables extreme parallel development without coordination bottlenecks. These are extremely strong for CD.

Databases in Continuous Delivery

In most systems, application relies in a single, centralized, monolithic database, shared by many modules, services and teams. This creates tight coupling across the entire system because numerous components depend on the same schema, tables and procedures. This contradicts the code principal of independent incremental, low-risk changes. To support Continuous Delivery, the database also must evolve alongside the code. This requires the database itself to become a deployable unit. Modern CD-friendly architectures achieve this by giving each component its own database. Each service owns its data and exposes access through APIs rather than direct table access. This means no foreign key across different databases. This approach eliminates cross-team coordination around database changes and prevents schema change for one feature from breaking unrelated areas of the system. Some other benefits are:

- Rollbacks are easier.
- Schemas are small.
- Radius of failures are smaller
- Teams have a clear ownership over data and reduces dependency.

Overall shift

The fundamental shift is from treating operations as a separable cleanup crew that receives finished software, to making production realities a central design constraint from day one. Instead of the linear sequence of ”design -¿ build -¿ handoff -¿ deploy -¿ operate”, we now design for deployment and operation from the start. This means architecture is shaped by how software will be monitored, scaled, secured, debugged and recovered in the live environment. Production isn’t a distant destination, it is the primary design criterion and operations isn’t a phase but a continuous concern baked into the very structure of the system.

Quality Attributes Continuous Delivery

Deployability

Deployability is the system's ability to be deployed or upgraded quickly, safely and reliably into production. In CD, this means:

- Every change is potentially shippable and can be deployed with minimal manual effort.
- Deployment is automated through a deployment pipeline that tests, validates and promotes artifacts.
- Rollbacks are as easy as deployments, minimizing risks.
-

Few set of signs of high deployability can be: you can deploy to production with one click, you can deploy during business hours without sweating and you can rollback a bad deployment within minutes.

Modifiability

Modifiability is crutial for CD because CD requires frequent, low-risk changes and high modifiability reduces the cost and risk of each deployment. While low coupling and high cohesions support modfiability, they are not the only factors. Here are some tactics that aid in maintaining high modifiability:
Tactic 1: Delayed Decisions - CD environments change constantly, so delay irreversible architectural decisions until you have sufficient information to choose wisely. Keep options open by using abstractions, interfaces and configurations rather than hard-coded choices.
Tactic 2: Stateless Architecture - The application itself shouldn't store any session/state inside of itself. Instead store it externally without any need for migration when a module of the application is upgraded/replaced. This lets you deploy/update/rollback individual components without affecting user sessions and removes a need for coordinate states when migrating across versions.
Tactic 3: Trunk-Based Development - In CD, you want to push your code to the main branch constantly. Never keep long-lived branches. If your feature isn't finished, tag & hide it in production. Long-lived branches can cause merge conflicts later as in CD, main branch keeps evolving.

Testability

In a CD pipeline, software goes through various automated testings at multiple stages, requiring the tests to be fast, reliable and maintainable. High testability enable rapid feedback and confident deployments. **Architectural Impact on Testability:**

- **Low Coupling**: Enables isolated unit testing without complex setup

- **Clear Interfaces**: Defines precise contracts for mock/stub implementations.
- **Separation of Concerns**: Allows testing business logic independently of infrastructure.
- **Deterministic Behavior**: Eliminates flaky tests caused by race conditions or external dependencies

Tactic 1: Design for the Test pyramid - Structure testing with many fast, isolated unit tests and fewer integration tests. This ensures quick feedback while maintaining coverage.
Tactic 2: Apply dependency Injection - Design components to receive dependencies such as databases, services and configurations as parameters rather than instantiating them internally. This enables inserting false data, in-place of production dependencies and allows for configuration of different behavior for different test scenarios.
Tactic 3: Establish Testing Contracts - Define explicit interfaces between components. This creates clear boundaries of what is required and not required to be tested.

Resilience

Resilience is a system's capacity to withstand failures and continue operating. Increasing the frequency of deployment increases the number and severity of failures. Failures are inevitable in a CD. CD doesn't aim to eliminate failures but instead, to make them cheap, contained and easy to recover from.
Why Resilience enables CD:

- Teams can deploy confidently knowing failures won't cascade.
- Mean Time to Recovery becomes more critical than Mean Times Between Failures.
- Contained failures enable faster rollbacks and diagnosis.

Reusability

Reusability in CD is more challenging than others. While shared components can accelerate development and ensure consistency, they can also create bottlenecks and coupling that hinder the process of CD. So, it is in best interest for an organization to find a balanced tradeoff with reusability and CD.
The Reusability Trade-off in CD:

- **Benefits**: Reduced duplication, consistent patterns, faster feature development, shared knowledge.
- **Risks**: Bottlenecks in shared teams, version conflicts, testing complexity, creates coupling deployment coordination.

- **Reality in CD**: Teams must deploy independently and shared components must not force synchronized releases.

Some reusability tactics that are CD-friendly are:
Tactic 1: Copy-paste over abstraction: Though typically seen as a bad practice, duplicating code rather than creating shared libraries is CD-friendly. "Duplication is cheaper than wrong abstraction" - Sandi Metz
Tactic 2: Design Reusable Services, not libraries - Instead of shared code library, create reusable microservices that allow teams to consume via well-defined APIs and not compile-time dependencies.