Comparison: Centralized vs. Distributed Version Control

1. **Architecture**:
   - **Centralized**: A single central server manages all versions.
   - **Distributed**: Each user has a complete local repository that mirrors the central repository.
2. **Example Tools**:
   - **Centralized**: Subversion (SVN), Perforce, CVS.
   - **Distributed**: Git, Mercurial, Bazaar.
3. **Dependency on Central Server**:
   - **Centralized**: Requires a constant connection to the central server for most operations.
   - **Distributed**: Work can be done offline, and synchronization occurs when needed.
4. **Performance**:
   - **Centralized**: Slower due to dependency on the central server.
   - **Distributed**: Faster as most operations are performed locally.
5. **Commit History**:
   - **Centralized**: Stored only on the central server; users do not have a local history.
   - **Distributed**: A complete commit history is stored locally on each user's machine.
6. **Conflict Resolution**:
   - **Centralized**: Conflicts are resolved on the central server.
   - **Distributed**: Conflicts are resolved locally before pushing changes to the remote repository.
7. **Branching and Merging**:
   - **Centralized**: Limited and less flexible; branches often directly affect the main codebase.
   - **Distributed**: Extensive and efficient; branches are isolated locally for independent development.
8. **Backup and Reliability**:
   - **Centralized**: High risk of data loss if the central server fails, as it is the single point of failure.
   - **Distributed**: Safer as multiple local copies ensure data redundancy and continuity.
9. **Scalability**:
   - **Centralized**: Performance can degrade with larger teams or repositories.
   - **Distributed**: Scales well with large teams and repositories.
10. **Flexibility**:
    - **Centralized**: Less flexible as users must adhere to the central server's workflow.
    - **Distributed**: Highly flexible; supports various workflows and decentralized collaboration.
11. **Examples in Use**:
    - **Centralized**: Commonly used in legacy systems or small teams.
    - **Distributed**: Widely adopted in modern projects, open-source development, and large teams.
12. **Collaboration**:
    - **Centralized**: Concurrent work is harder due to centralized control.
    - **Distributed**: Easier collaboration with robust branching and merging tools.
13. **Reliability**:
    - **Centralized**: Prone to failure; if the central server goes down, work cannot proceed.
    - **Distributed**: More reliable since local repositories allow uninterrupted work.

This comparison highlights how **centralized systems** focus on simplicity and control, whereas **distributed systems** prioritize flexibility, performance, and reliability.

Key Benefits:

1. **Easy Rollback:** Users can quickly be routed back to the stable version if issues arise.
2. **Feature Usage Insights:** Allows companies to assess user engagement and revenue impact of new features and roll back if necessary.
3. **Capacity Testing:** Gradually increases the load on the application, measuring metrics like CPU, memory, and response time, ensuring it meets capacity requirements.

This approach enables safe releases with minimal risk and avoids the need for a large capacity testing environment.

Deploying and Promoting Application

FIRST DEPLOYMENT

The first deployment focuses on automating the process of building, testing, and deploying the app to an environment that closely matches production. It ensures consistency, checks basic functionality with simple tests, and makes sure the setup in development mirrors the production system. The goal is to make deployments reliable and repeatable.

Modelling Release Process and Promotion of Builds:

As the application grows, deployment becomes more complex. The release process involves several stages like Integration Testing, Quality Assessment, and Staging. Each stage has approval gates with specific authorities. The build is promoted through these stages, allowing for easy testing, self-service deployments, and demos. The workflow includes selecting the build, preparing the environment, deploying the app, configuring it, testing it, and promoting it to the next stage if successful. If not, the issues are recorded for improvement. The process ensures consistent and automated deployment across environments.

Promoting Configuration:

In addition to the application binaries, its configuration and environment must also be promoted. This involves ensuring that new configuration settings work as expected, verified by tests like a Smoke test. For instance, checking if a string returned by an external service matches the environment before deployment. Middleware settings like thread pools can be monitored using tools like Nagios. In service-oriented or componentized architectures, all services or components must be promoted together to ensure consistent functionality across the application.

Orchestration

Orchestration is about managing multiple applications running in the same environment without causing conflicts. When different applications or versions of the same application share an environment, careful attention is needed to prevent disruptions. Virtualization technology can help by isolating each application. When applications need to interact, especially in service-oriented architectures, integration testing ensures they work together. A smoke test is often used to check if everything is functioning correctly. In short, orchestration deals with coordinating releases that involve multiple systems working together.

Deployments to Staging Environments:

Before releasing an app, it’s tested in a staging environment similar to production. Key steps include preparing the environment, automating configurations, performing smoke tests, and testing integration with external systems. Techniques like Blue-Green and Canary releasing ensure smooth transitions to production. Always deploy to staging first before going live.