2. Source control strategy

└ 2.3 Package Management

└ 2.3.3 Versioning strategies: SemVer, CalVer

---

1. What is Semantic Versioning (SemVer) and what are its components?
2. What is Calendar Versioning (CalVer) and how is it structured?
3. When should you use SemVer versus CalVer?
4. How do pre-release versions and build metadata work in SemVer?
5. How can you implement SemVer in CI/CD pipelines?
6. How do you implement CalVer in package versioning?
7. How does versioning affect dependency management?
8. How can you enforce versioning rules in Azure DevOps or GitHub pipelines?
9. What are best practices for updating package versions?
10. How can you handle breaking changes in a versioning strategy?

---

1. **What is Semantic Versioning (SemVer) and what are its components?**
   SemVer is a versioning scheme with three numeric components: MAJOR.MINOR.PATCH (e.g., 2.1.3).

- MAJOR: Breaking changes
- MINOR: Backward-compatible features
- PATCH: Backward-compatible bug fixes

---

2. **What is Calendar Versioning (CalVer) and how is it structured?**

CalVer is a date-based versioning scheme, typically using formats like YYYY.MM or YY.MM.DD (e.g., 2024.06 or 24.06.05), representing the release date.

---

3. **When should you use SemVer versus CalVer?**
   - Use *SemVer* when precise tracking of breaking and backward-compatible changes is needed.
   - Use *CalVer* where the release schedule is more important than feature tracking.

---

4. **How do pre-release versions and build metadata work in SemVer?**
   - Pre-release versions use a hyphen and identifier (e.g., `1.2.0-beta`).
   - Build metadata uses a plus sign (e.g., `1.2.0+001`).
   - Pre-release versions are lower precedence than final releases.

---

5. **How can you implement SemVer in CI/CD pipelines?**

Automate version bumping based on commit messages or tags, validate version format, and update package files in the build pipeline before publishing artifacts.

---

6. **How do you implement CalVer in package versioning?**

Set the version number in build scripts or pipeline templates to the current date (e.g., using a variable or script to generate YYYY.MM.DD) during the build process.

---

7. **How does versioning affect dependency management?**

Correct versioning allows tools to resolve compatible dependencies, prevent breaking changes, and manage upgrades/downgrades reliably across services and packages.

8. **How can you enforce versioning rules in Azure DevOps or GitHub pipelines?**
    - Add validation steps to CI/CD pipelines that check version format
    - Enforce increment rules
    - Reject builds if versioning is not compliant (using custom scripts or tasks)

---

9. **What are best practices for updating package versions?**
    - Update versions before publishing,
    - Follow a consistent versioning policy,
    - Document changes,
    - Communicate breaking changes clearly.
    - Automate version updates in pipelines when possible.

---

10. **How can you handle breaking changes in a versioning strategy?**
    - Increment the MAJOR version in SemVer, document breaking changes in release notes, and communicate impacts to consumers.
    - In CalVer, document breaking changes in changelogs since the version number alone doesn't indicate compatibility.