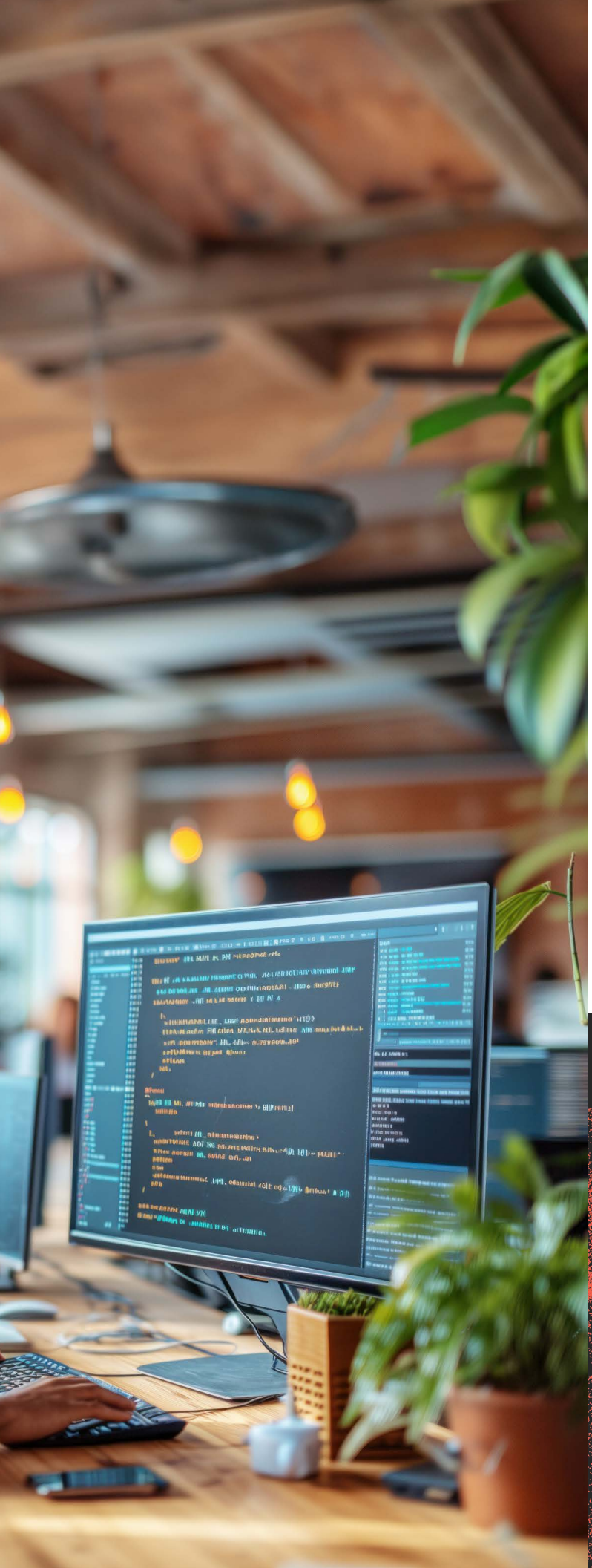


LTM

WHITEPAPER

Navigating the Angular Upgrade Maze: Challenges, and Their Strategic Solutions

Lakshminarayanan Ulaganathan
Solution Architect

N Prabakaran
Solution Architect

Karuppaiah Palaniappan
Solution Architect

Vamsi Dasari
Tech Lead

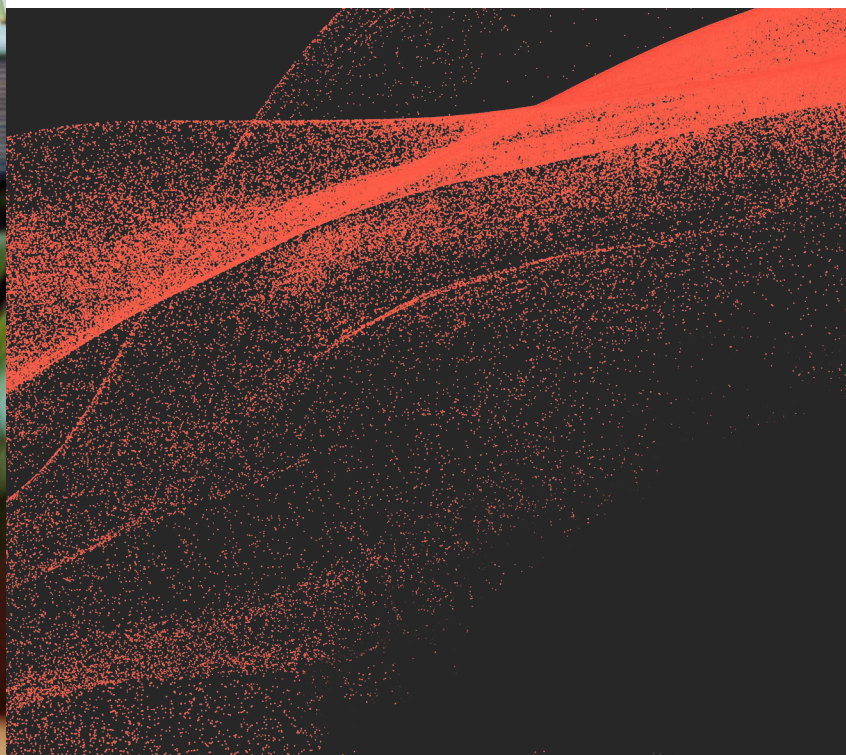


Table of Contents

A practical look at why Angular upgrades become complex in enterprise environments and how teams can Outcreate possibilities by handling dependency risks, breaking changes, and modernization decisions with clarity and control.

Executive Summary

Angular remains a strong choice for enterprise applications because it offers structure, consistency, and long-term maintainability. Yet in real delivery environments, upgrading Angular rarely stays limited to the framework itself. What begins as a routine version change often expands into a broader exercise that touches third-party packages, UI libraries, TypeScript versions, build tools, testing cycles, and release timelines. That is where many engineering teams lose time, confidence, and momentum.

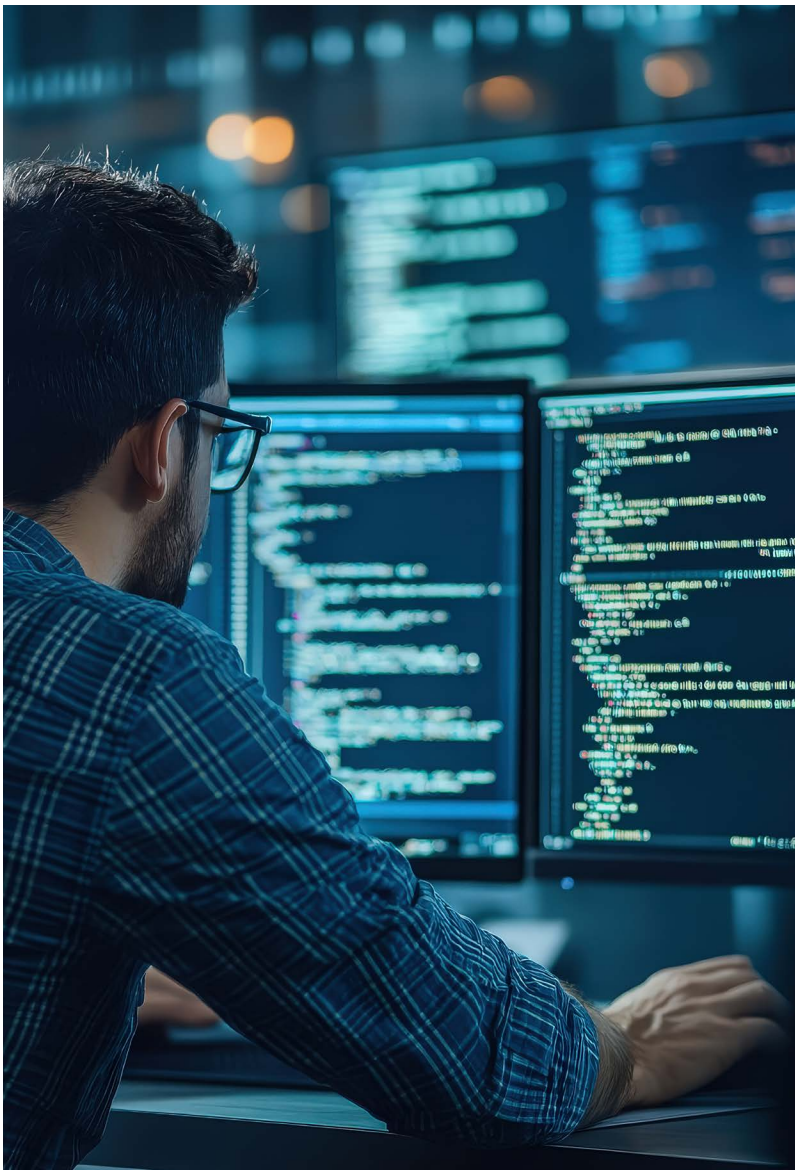
This whitepaper examines why Angular upgrades can become disproportionately difficult in enterprise settings, even for mature teams with well-architected applications. It draws on our experience maintaining multiple homegrown platforms built on Angular, where upgrade decisions have repeatedly involved more than just code changes. In practice, one dependency update, one security advisory, or one ecosystem shift can trigger a much larger chain of engineering and quality assurance effort than teams initially expect.

The paper focuses on four recurring realities:

- External package dependencies often force Angular upgrades earlier than product roadmaps would prefer.
- The real effort often sits in the surrounding ecosystem and not in the Angular core.
- Teams frequently upgrade to stay supported but delay adoption of newer Angular patterns, which increases technical debt over time.
- Breaking changes amplify the effort across development and testing, especially in large applications with wide functional coverage.

This paper is intended for engineering leaders, architects, platform owners, and delivery teams who need to make Angular modernization decisions with greater clarity. It is especially relevant now because Angular's release cadence, evolving architecture, and tightly linked dependency ecosystem continue to raise the cost of reactive upgrades.

Rather than treating upgrades as routine maintenance, this paper argues for a more strategic approach. Teams need to plan upgrades as part of broader platform modernization, dependency governance, and engineering quality. That shift helps them reduce disruption, protect delivery velocity, and make modernization choices that support long-term outcomes.



Introduction

Angular, developed and maintained by Google, remains one of the most widely used frameworks for building scalable, enterprise-grade applications. Its strong architecture, opinionated structure, TypeScript-first approach, and mature tooling make it a dependable choice for organizations that value consistency and maintainability across large application landscapes.

At the same time, upgrading Angular in an enterprise environment is rarely simple. The challenge goes beyond moving from one framework version to another. In our experience, maintaining a suite of homegrown Angular platforms, even a seemingly contained upgrade, can expand quickly once peer dependencies, security patches, UI libraries, build tools, and regression cycles come into play. A change that looks minor on paper can turn into a multi-team effort with real delivery impact.

Angular's release cadence adds to that pressure. With major releases every six months and limited support windows, teams cannot afford to stay behind for long. Delayed upgrades increase exposure to unsupported versions, security concerns, tooling gaps, and rising effort when the next change finally becomes unavoidable.

This whitepaper explores the common realities of Angular version upgrades and the practical challenges teams face as they navigate them. It examines dependency lock-in, ecosystem-driven complexity, underuse of newer Angular capabilities, and the testing burden caused by breaking changes across the stack.

The intent is simple: to help engineering teams approach Angular upgrades with more foresight, stronger governance, and a clearer modernization path. Because in large enterprise applications, upgrade decisions do not affect only code health. They shape delivery speed, platform resilience, and the ability to keep building with confidence.

Version Upgrades: Why They Feel Like "Upgrade Hell"

Upgrading to a new version of any software is imperative. Upgrades usually bring in new features, improve performance or usability, address issues, and address security vulnerabilities. Unlike libraries that can often be upgraded incrementally, Angular version upgrades can have cascading effects. Each upgrade typically includes framework changes, RxJS updates, TypeScript version shifts, CLI changes, Webpack adjustments, and updates to a long list of peer dependencies. Additionally,

- Angular maintains Active LTS for 6 months and Extended LTS for 12 months, making timely upgrades mandatory.
- Older versions become unsupported relatively quickly, exposing teams to security vulnerabilities, outdated tooling, and a lack of community help.
- Real-world applications rely on numerous third-party libraries, many of which lag behind Angular's fast release cadence.

As the team completes a version upgrade and goes live, a new version of Angular has already been released. As a result, engineering teams often end up in a constant race: updating Angular and its dependencies, resolving breaking changes – all while ensuring application stability. This sustained maintenance burden diverts valuable engineering capacity from feature development and business-driven innovation to technical debt.

Key Challenges Faced During Version Upgrades

Below are the most common issues we encountered during Angular version upgrades, along with examples and potential solutions to overcome them.

Dependency Lock-In: When External Packages Force Your Hand

Even if your team does not plan to or does not want to upgrade Angular, third-party NPM packages often force an upgrade indirectly.

Example:

A team using Angular v15.x and Keycloak v21.x as their IAM solution may discover a high-severity vulnerability in the version and need a keycloak version upgrade to Keycloak v25.x to address the issue. The patched version of the library, however, supports only Angular 16+, while the application is still on Angular 15. This creates a dependency lock in situation:

- Security team mandates the upgrade.
- Developers cannot upgrade the package without upgrading Angular.
- Angular upgrade requires multiple other dependency upgrades.
- Although upgrading Keycloak requires minimal effort, updating Angular is a much more challenging task.
- A simple patch snowballs into a multi-week effort.

This creates frustration because external packages dictate the upgrade timeline, not your product roadmap.

Solution

A solid governance approach to dependency management helps in being in the driver's seat of upgrades rather than being dictated a plan to upgrade. The following steps help in reaching there:

- Maintaining a simple dependency compatibility matrix for your app.
- Keeping a watch on the security vulnerabilities and upgrade cycle of the key dependencies (e.g., Keycloak, Chart.js etc.,) of your app:
- Subscribing to security advisories – newsletters and blogs.
- Regular monitoring of blogs of Angular and the critical dependencies.
- Follow strict version pinning to avoid unexpected auto-upgrade of minor version changes:
- Consciously defining the SemVer (Semantic Version) for each dependency. “keycloak-js”: “22.0.5” is stricter than “keycloak-js”: “^22.0.5”.

How can AI assist? AI tools can help engineering teams proactively identify potential security vulnerabilities in third-party packages and suggest appropriate version pinning.

The Real Upgrade Pain Isn't Angular – It's the Ecosystem

Framework upgrades themselves (e.g., running ng update @angular/core) are often manageable.

The real challenge lies in upgrading the dependent package ecosystem.

Libraries such as PrimeNG, RxJS, Node runtimes, and many more often have strict peer dependency requirements, breaking changes, or delayed support for new Angular versions.

Example:

PrimeNG, one of the most widely used UI component libraries, frequently introduces breaking CSS and component-level changes across major versions. This requires:

- Template rewrites.
- Styling updates.
- Behaviour adjustments.
- Regression testing.

The dropdown component was renamed from '<p-dropdown>' to '<p-select>' in PrimeNG 17 to 18 version upgrade, forcing teams to update templates across the application.

In many cases, upgrading a third-party library like PrimeNG becomes more challenging and time-consuming than upgrading Angular itself, because the impact is widespread and often requires manual intervention.

Solution

Carefully evaluate and select the major external dependencies for your app. From a UI component & theming framework perspective, the two main contenders are Angular Material and PrimeNG. While both frameworks have introduced breaking changes in major version upgrades, Angular Material provides better automation support for upgrades via CLI –

ng update @angular/material@21. While this is not a magic command to finish the upgrade in one click, it saves a significant amount of developer effort during the upgrade process. So, pick the right tools from the ecosystem, evaluating their active development and patch release cycle, and their history of breaking changes during upgrades.

How can AI assist? While handling this kind of breaking changes involves a lot of manual repetitive activities, this is where an AI tool like GitHub Copilot can provide a helping hand and do the heavy lifting:

- Provide a precise context of the version upgrade to Copilot.
- Write a clear to-the-point prompt to get a strong regex patten to find-and-replace the component '<p-dropdown>' to '<p-select>' covering all edge cases.
- Even further, use the Agent Mode of Copilot to perform the find-and-replace activity across the code base.

Although using an ecosystem is inevitable, its scope can be reduced by avoiding unnecessary third-party libraries. Many trivial dependencies, especially simple UI helpers, add more long-term cost than value. For example, teams frequently install lightweight packages such as ngx-spinner just to show loading indicators, even though Angular already provides the necessary tools to implement this with a small custom component. By building such simple custom components in-house, teams reduce external dependency surface area and retain more control over their upgrade timeline.

Underutilization of New Angular Features

Many teams upgrade Angular "just enough to stay supported" without adopting new features that can simplify the codebase. This leads to technical debt of "Upgrade but don't evolve": you're on the latest version but not leveraging improvements that could reduce future upgrade effort. At times, this might lead to a significant code rewrite during future upgrades when Angular deprecates or retires a feature that was discouraged a few versions earlier.

Example:

Angular introduced Standalone Components in version 14. However, many teams that upgraded to Angular 14, 15, or 16 continued to utilize NgModule extensively. Although NgModule remained the default approach until Angular v16, starting with Angular v17, standalone components became the default, while support for NgModule persisted. Numerous engineering teams prefer to set standalone = false and continue using NgModule. This means:

- More boilerplate code.
- Missed performance benefits.
- Tougher future upgrades.

Solution

Addressing this issue primarily involves cultivating the appropriate mindset within the engineering team, rather than categorizing it as technical debt. The team should be encouraged to leverage the latest enhancements of the framework and proactively seek opportunities to create business value through their adoption. The most effective way to avoid "upgrade only" technical debt is to adopt new Angular features incrementally rather than waiting until a major rewrite becomes unavoidable.

Teams can:

- Deliberate and pick the most valuable enhancement.
- Start by enabling new patterns only in new features or modules.
- Moving to a new pattern when making changes to an existing module and thereby evolving to the entire app.

How can AI assist? AI tools can greatly shorten developers' learning curves, helping them understand new features and providing recommendations on how and where to use them effectively.

Breaking Changes Amplify Effort Across Dev & QA

Almost every Angular release comes with some form of breaking change. Combined with breaking changes from the ecosystem — PrimeNG, RxJS, or TypeScript — this translates into large-scale refactoring, regression across multiple modules, and significant testing cycles. In large applications with wide functional coverage, this disruption is costly and unavoidable without strong upgrade discipline.

Example

A typical major version upgrade can trigger a chain of issues across the codebase:

- Layout breakages due to theme (e.g., PrimeNG) updates.
- Incompatible TypeScript signatures.
- Lifecycle hook deprecations.
- API changes in libraries.

The ripple effect results in:

- Extensive development effort.
- Significant regression testing cycles.
- Potential production issues if not thoroughly validated.

Solution

When an upgrade is inevitable, automation plays a key role in reducing the effort required. Automated tests for functionality and UI using tools like Playwright or Cypress can help identify issues early in the upgrade cycle, save considerable time, and avoid post-Go-Live issues.

How can AI assist?

AI not only supports the upgrade process but also helps generate automated tests in your preferred tool. Creating these tests incrementally for the whole app boosts overall quality, benefiting the app beyond the upgrade cycle.

Conclusion: Time to Think Beyond Angular?

Angular continues to be a powerful, opinionated, enterprise grade framework with strong architecture. However, its frequent major releases, heavy dependency ecosystem, and tightly coupled structure raise a valid strategic question:

Is Angular still the right long-term stack for your organization?

React – one of Angular's closest alternatives- offers a contrasting value proposition:

- An incremental upgrade model with far fewer breaking changes.
- Lightweight and flexible architecture.
- A vast ecosystem of community and enterprise tooling.
- Much better LTS period.
- Freedom to adopt only the libraries you need.

Yet React is not without drawbacks. Here is a quick comparison of Angular vs React on various dimensions

Dimension	Angular	React
Ownership	Google	Meta
Type	Full framework	UI library
Architecture	Strict, structured	Flexible, customizable
Learning Curve	Steep	Moderate
Upgrade Experience	Frequent breaking changes	Mostly incremental
Ecosystem Coupling	Highly coupled	Loosely coupled
Release Cadence	Fast, predictable	Slow, stable
UI Libraries	Stable (Material), volatile (PrimeNG)	Broad, mature options
Performance	Great for large apps	Great for interactive UIs
Talent Pool	Smaller	Very large
Maintainability	High with strict discipline	High with lower upgrade cost
Risk Profile	Higher upgrade risk	Lower upgrade risk
Security	Built in security features (XSS, sanitization, templates)	Relies more on developer practices + external libraries

Other modern options, such as Vue, Svelte, or Astro, continue to evolve rapidly, offering compelling performance benefits, simpler mental models, and more ergonomic developer experiences. But these come with their own adoption curves and risks.

Ultimately, choosing the right front-end framework is a strategic investment decision. Organizations must evaluate:

- Engineering team maturity.
- Project complexity and longevity.
- Upgrade budgets and release pressure.
- Long-term support expectations.
- Alignment with broader architectural direction (micro-frontends, design systems, API-first models, etc).

Given the complexities of Angular version upgrades and our engineering team's strategic objectives, we are inclined toward React.

About the Authors

Lakshminarayanan Ulaganathan
Solution Architect

Solution Architect skilled in collaborating with business & engineering teams to design effective, scalable solutions. Passionate about solving business problems through a user-first mindset, focusing on understanding end-user pain points and applying the right technology to deliver practical, high-impact outcomes.

N Prabakaran
Solution Architect

Technical Architect with strong full-stack experience, specializing in scalable web applications. Skilled in Angular and React, with expertise in migration strategies, microservices, and cloud-native solutions. Proficient in backend technologies including Java, .NET, and Python. Experienced in leveraging AI tools like Claude, Copilot, and Agentic AI to enhance development, productivity, and code quality.

Karuppaiah Palaniappan
Solution Architect

Solutions Architect specialized in high-impact EHS platforms for Construction and Oil & Gas. Experienced in orchestrating AWS and Azure integrations and architecting scalable, real-time worker tracking solutions to enhance site safety. Driven by engineering efficiency, developed reusable cloud-native components that reduce technical debt and ensure seamless interoperability across complex, mission-critical industrial ecosystems.

About the Authors

Vamsi Dasari

Tech Lead

A seasoned Tech Lead and front-end specialist with deep expertise in Angular and React, bringing a strong focus on building scalable, high-performance web applications. Driven by modern UI architecture and user experience best practices, this leader enables teams to deliver robust and maintainable front-end solutions.

References

1. *Angular versioning and releases,* Angular: https://angular.dev/reference/releases#release-frequency
2. *Versioning Policy,* React: https://react.dev/community/versioning-policy
3. *CVE Record: CVE-2024-1132, CVE.org,* April 17, 2024: https://www.cve.org/CVERecord?id=CVE-2024-1132
4. *keycloak-angular, npm,* December 31, 2025: https://www.npmjs.com/package/keycloak-angular
5. *Long Term Support,* PrimeNG: https://primeng.org/lts