International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 5, Issue 1, 222-231, 2024
ISSN: 3050-9262 | httpss://doi.org/10.63282/3050-9262.IJAIDSML-V5I1P123

Original Article

AOT vs JIT Compilation in Ivy: Benchmarks & Trade-offs

Kavya Muppaneni
Software Engineer at HCL Global Systems, USA.

Abstract - *Angular's Ivy rendering engine is a major change of the framework's compilation and rendering paradigm. Essentially, it changes the way the older View Engine works by replacing it with a more efficient, instruction-based architecture that allows locality-aware compilation, better tree shaking, and smaller, more predictable output. In this case, comparing AOT with JIT is very important for understanding how Ivy affects developer experience and production performance. This article compares AOT and JIT to real-world scenarios through benchmarks, which shows that bundle size, build time, startup latency, and runtime are some of the metrics that can be measured. The results indicate that Ivy-powered AOT is always better because it provides smaller bundles and faster load times, thus production deployment is the most favorable scenario. On the other hand, JIT retains its position during development because of fast rebuild cycles and template compilation on-the-fly, however, at the cost of increased runtime. The research breaks down Angular Ivy innovations in architecture to give a clear understanding of the results, showing how the compilation locality and tree-shakable instructions affect the performance trade-offs in different environments. The contributions embody a thorough empirical comparison of AOT and JIT under Angular Ivy, architectural analysis of internal mechanisms leading to differences in performance observable, recommendations for deployment of the engineering teams, and investigation of next steps like better incremental builds and hybrid compilation workflows all aimed at helping determine the best compilation strategy for modern Angular applications.*

Keywords - *Angular Ivy, Ahead-of-Time Compilation, Just-in-Time Compilation, Angular Compilation Pipeline, Tree-Shaking, Bundle Optimization, Web Performance, Build-Time Optimization, Runtime Performance.*

1. Introduction

Modern front-end development has changed on a very large scale within a short time. Applications are no longer just simple groups of static pages but are dynamic, feature-rich single-page applications (SPAs) that are expected to provide almost native performance across devices and network conditions. While frameworks like Angular, React, and Vue are competing to offer advanced rendering capabilities, the changes in the underlying compilation strategies have become the main factor to both efficient development workflows and optimized end-user experiences. With the Ivy engine, Angular has moved to a major architectural overhaul and is fundamentally changing the way Angular compiles, generates, and executes application code. In fact, with such a change, the old debate between Ahead-of-Time (AOT) and Just-in-Time (JIT) compilation should now be reconsidered based on Ivy's design principles, capabilities, and limitations.

1.1. Challenges in Modern Front-end Compilation

The substantial behavior of SPAs overtly suggests that frontend applications are currently the main processors of the state management, routing, forms, internationalization, data orchestration, and complex UI interactions, etc. all within the browser. Such a complicated issue raises the level of the compilation pipelines' burden that has to go on to produce a highly optimized output capable of efficiently running on different environments. Thus bundle sizes are still among the most stubborn problems of the JavaScript world: the bigger the bundle is, the slower the first load time is, the memory usage is higher, and the CPU cost is increased during parsing and execution. Whereas these problems are even more severe on low-power mobile devices which comprise a significant portion of the global web traffic.

At the same time, developers demand quick feedback loops instant recompilation, hot module replacement, and rapid rebuild cycles to keep up their productivity. Balancing this developer experience (DX) with production-level performance creates an invisible border between different compilation strategies. Even though JIT is generally used for local development and is faster because templates are compiled directly in the browser, AOT has the advantage of runtime efficiency as templates are precompiled, and tree-shaken. With the growing demand for continuous delivery and tighter CI/CD integration, the dilemma becomes even more complicated as compilation strategies have to be in line with automated build pipelines, testing workflows, and deployment environments.

Mobile-first design principles and the worldwide trend towards bandwidth-limited users are factors that force frameworks to put more emphasis on smaller bundles, which are less resource-consuming, and faster startup performance. As more and more companies are targeting emerging markets and low-connectivity regions, compilation decisions should take into account not only developers' preferences but also constraints. These constraints are a collective call that modern front-end compilation

is a matter of deep understanding, and must have the right set of tools and empirical evaluation, particularly in the case of Ivy's new architecture.

1.2. Problem Statement

Even with Ivy's improvements, many teams are still confused about whether they should use AOT or JIT. In the past, the differences between AOT and JIT in different Angular versions were quite clear AOT would take more time to build but the app would be fast, while JIT would allow fast development but the app would be slower at runtime. Ivy changes these characteristics because of its locality-based compilation model, tree-shakable instruction sets, and a rendering pipeline that is quite different. However, the performance differences that exist in the real world and are mirrored in the Ivy ecosystem have not been fully explained by the remaining documentation, community benchmarks, and Angular updates.

There are still quite a few critical questions that are not sufficiently answered: What is the comparison of build sizes between Ivy AOT and JIT? What are the actual runtime performance differences in rendering, change detection, and bootstrapping? How do compilation times vary between development, testing, and CI environments? What influence does Ivy exert on hydration, SSR compatibility, or features like partial hydration in the future? In addition to this, the teams are not sure how the mode of compilation affects the incremental builds, whether the Angular CLI configurations are impacted, and what effect is there on the emerging patterns like standalone components. Without a set of systematic benchmarks that are specific to Ivy, developers cannot but fall back on outdated assumptions and anecdotal pieces of advice, thus building suboptimal strategies for development and deployment.

1.3. Motivation

This research is motivated by the major changes Ivy brings in the way Angular compiles under the hood. Since Ivy can do a fine-grained compilation, produce a more compact output, and allow advanced tree shaking, the traditional understanding of AOT versus JIT is not valid anymore and thus needs a data-driven solution. Developers want a single view that performance metrics, DX considerations, architectural insights, and deployment implications will integrate into one decision-making framework.

The most recent study comparing AOT and JIT with Ivy is a timely response to the demands of companies that progressively include evidence-based optimizations as a primary part of their CI/CD pipelines, in performance-sensitive or large-scale applications, mainly, where small inefficiencies could have a big impact. The knowledge of different compilation modes influence on build reproducibility, caching strategies, test runtimes, and deployment artifacts can significantly affect costs, reliability, and the user experience. Angular's rapid move to standalone components, better SSR, and Ivy features for the future only increases the need for updated, complete benchmarks. This study gives a way to teams to understand the evolution of the Angular ecosystem with clarity and confidence.

2. Literature Review

2.1. Background on Angular Compilation

Before Ivy, Angular depended on the View Engine, which was a compilation and rendering system that uses a centralized metadata graph to transform Angular templates into generated code. The View Engine needed the metadata of the whole application to be checked during compilation, thus the builds were slow, there were few tree-shaking possibilities, and the factory files were of a large size. Template expressions were turned into big factories that encoded structural and attribute instructions in a verbose way, thus the bundles were big, and the flexibility was limited. The arrival of Ivy was a major architectural change. With Ivy's local compilation model, components can be compiled separately, thus a monolithic metadata graph is not needed, and incremental builds can be done faster. Its rendering pipeline is derived from an Incremental DOM-like approach, where instructions correspond to discrete DOM operations. This instruction-set-driven mechanism not only keeps the size of the generated code to be very small but also makes the runtime faster as it can do very granular DOM updates. Ivy takes the elimination of the dead code one step further by organizing the generated code in very modular instruction sets so that the unused features can be removed much more efficiently. In general, these changes have adjusted Angular's compilation strategy to the present time and have had an effect on how AOT and JIT operate in the new pipeline.

2.2. AOT Compilation in the Literature

Ahead-of-Time (AOT) compilation has generally been advised for production builds because of its various performance advantages. The resources mention its benefits in the View Engine era, i.e., faster startup time, smaller bundles, and more predictable runtime. Besides that, AOT's security advantages are also very clear; as templates are precompiled into TypeScript and JavaScript instructions rather than being parsed dynamically in the browser, the possibility of template injection or runtime expression execution which are often heavily linked to XSS vulnerabilities is minimal. Most of the time, research done before Ivy showed that AOT had better runtime performance with studies reporting improvements in initial load performance and elimination of costly client-side template parsing. Nevertheless, these publications mostly considered AOT under versions of Angular that did not have locality, instruction-based rendering, and better optimization strategies brought by Ivy.

Consequently, the performance assumptions about AOT should be reconsiderated in the case of the rearchitected engine of Angular.

2.3. JIT Compilation in the Literature

Throughout the years, Just-in-Time (JIT) compilation has been generally chosen in dev mode as it gives the possibility for templates to be compiled on the client side, hence rapid trial and getting the rebuild time down. The research before, mentions JIT as a means for fast feedback loops which is very important for big or fast changing applications. Yet, they also point out its several inherent limitations in their respective researches. JIT builds usually result in bigger bundles because templates are in a source-like form so are the JIT build files, leading to longer loading times and higher memory consumption. In addition, runtime template parsing may cause a heavy overhead and thus the startup performance is sometimes considerably slower than in the case of AOT. JIT's runtime character also limits the possibilities of certain optimizations dependent on static analysis, e.g. more drastically tree-shaking or precomputation of static values. These discoveries reflect the worthy insights, however, they refer to the behaviors associated with the compilation pipeline that was before Angular Ivy and do not indicate fully how JIT co-operates with Ivy's rendering model.

2.4. Ivy-Specific Studies

Many studies and community analysis have been done after Ivy's introduction in an attempt to understand its redesigned build pipeline which is largely focused on the benefits of locality-based compilation, factory sizes reduction, and uniformity improvement across compilation modes. These studies indicate that Ivy is capable of generating smaller instructions and component definitions, thus enabling more efficient code elimination and the reduction of the size difference between compilation modes. The studies document that Ivy architectural characteristics like the instruction-driven renderer and incremental compilation are the main factors that lead to faster builds and more performant runtime execution. Nevertheless, existing research uncovers huge gaps. Only a handful of studies have been performed, which, in an Ivy-only context, directly compare AOT versus JIT; and a very small number of studies examine performance differences in realistic, production-grade applications. Moreover, the literature review is devoid of systematic benchmarking that thoroughly considers behavior specific to Ivy such as instruction generation, build caching, and locality-driven optimizations.

2.5. Summary of Research Gap

Most of the research mentioned is based on assumptions from the View Engine era and it is quite evident that there is a disconnect in the way AOT and JIT are understood under Ivy's radically changed compilation model. The previous benchmarks do not represent the changes made to Ivy nor the increasing use of standalone components and advanced SSR features. Consequently, there is an urgent demand for current empirical studies which look at both micro-level (component-level, instruction-level) benchmarks and macro-level (application-scale, CI/CD-integrated) scenarios. This study moves toward closing that gap by offering a contemporary, Ivy-centric comparison of AOT and JIT compilation.

3. Proposed Methodology

3.1. Research Design

This research employs a comparative experimental approach to determine the differences in the performance of Ahead-of-Time (AOT) compilation and Just-in-Time (JIT) compilation in Angular's Ivy environment. As Ivy changes the entire way in which the compilation is done locality-based compilation, instruction-based rendering, and improved tree shaking—any standard assumptions about AOT vs JIT performance have to be first verified by controlled, reproducible experiments. Hence, the methodology is about measuring behaviors during the build and runtime stages through various application sizes and configurations.

Five main groups of performance metrics have been defined to cover the whole range of differences:

- Build-Time Metrics — these are the time measurement of initial builds, rebuilds, and incremental compilations, thus, depicting the developer experience and CI/CD compatibility.
- Bundle-Size Metrics —these measure the size of the final JavaScript bundles, which may consist of main, vendor, and polyfill bundles, thus showing the effect of the compilation mode on the network transfer overhead.
- Runtime Performance — Metrics like startup latency, script evaluation cost, and frame rendering stability aim at making visible the performance differences of the end users.
- Memory Consumption — Attention is paid to both browser-side memory footprint and Node-side memory usage during compilation in order to grasp the resource efficiency.
- CPU Usage — The CPU impact during the build-time (Node) as well as the runtime (browser) stages is a factor in determining the computational demands of AOT vs JIT.
- Load-Time Metrics — Core Web Vitals, for instance, Time to Interactive (TTI), First Input Delay (FID), and Largest Contentful Paint (LCP), have been taken into consideration in order to associate the compilation strategy with the user experience.

3.2. Test Application Setup

In order to achieve representative and scalable results, a set of three standardized Angular Ivy applications will be developed. These applications vary in terms of their size, architectural complexity, and feature composition, thereby allowing performance measurements to be made in a wide range of realistic scenarios.

3.2.1. Small-Scale Application

- Around 10 components
- Simple routing (3–4 routes)
- Minimal or no external libraries
- No SSR, lightweight services
- Just right for micro-benchmarks and baseline comparisons
- Medium-Scale Application

3.2.2. Approximately 80 components

- State management (e.g., NgRx or signals-based stores) is used
- Lazy-loaded modules or standalone route-based lazy loading
- Moderate use of forms, dynamic templates, and HTTP data
- A typical mid-sized production Angular app is reflected
- Large-Scale, Enterprise-Level Application
- 200+ components with deep nesting
- Server-side rendering (SSR) enabled using Angular Universal
- Complex forms, multi-step workflows, real-time data synchronization
- Third-party libraries (charts, maps, virtualization, etc.)
- Authentication, role-based routing, and caching strategies
- Represents high-performance and scalability needs in enterprise environments

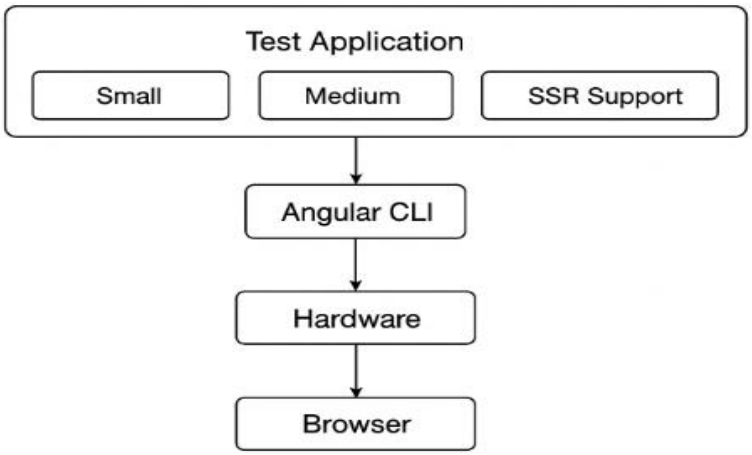

Fig 1: Experimental Setup Architecture

Each application will be created with Angular CLI version X (you can indicate the latest stable, e.g., v17 or v18), that is Ivy is the default compilation engine. Besides that, the research involves a comparison of the isolated component structure with the traditional NgModule-based way. Since Ivy is best suited for standalone components, this difference may have an impact on both AOT and JIT performance characteristics.

3.3. Experimental Environment

In order to stay consistent with the benchmarking, it is absolutely necessary to have a controlled experimental environment. The hardware and software specifications mentioned below will be the standard from which other setups will be compared:

3.3.1. Hardware Environment

- CPU: Quad-core or 8-core processor (e.g., Intel i7/Apple M-series/AMD Ryzen)
- RAM: 16–32 GB
- Storage: SSD-based machine

- Operating System: Windows 11 / macOS / Linux (use one for primary tests; others for cross-validation)

3.3.2. Software Environment

- Node.js LTS version (e.g., v20.x)
- Angular CLI vX with Ivy renderer enabled automatically
- Package manager: npm, yarn, or pnpm (the same one for all experiments)
- Build toolchain: Angular CLI default (Webpack or Esbuild depending on Angular version)
- Execution Environments
- Desktop Browser Testing: Latest Chrome (stable)
- Mobile Device Simulation: DevTools throttling profiles for CPU, network, and RAM
- Optional real-device testing on Android/iOS for hydration and runtime verification

All these conditions are the environment in which the builds and runtime experiments will take place and be consistent.

3.4. Benchmarking Parameters

Benchmarking is organized into the following categories:

3.4.1. Build Time

- Initial Build Time: Time from execution to the compilation being complete for the case of AOT vs JIT.
- Rebuild Time: The activations of changes and the measuring of incremental build performance.
- Cold vs Warm Cache Builds: The purpose of this is to determine Angular's build caching and Ivy's locality impact.

3.4.2. Bundle Size

- Main Bundle: The essential logic of the app.
- Vendor Bundle: The dependency and the Angular framework code.
- Polyfills Bundle: The different browsers compatibility layers.
- Source maps: an optional addition to track the effect on the build artifacts.

Bundle-size comparisons are a primary measure of network efficiency and initial load overhead.

3.4.3.. Runtime Metrics

- Time to Interactive (TTI): It is the measure of the shortest time user interaction is possible.
- Time to First Paint (TTFP) & First Contentful Paint (FCP): Indicative of the initial rendering.
- Largest Contentful Paint (LCP): Core Web Vital for the time visually coming loaded.
- Script Evaluation Time: The cost of parsing/executing JavaScript bundles.
- Memory Footprint: The usage of the heap and the frequency of garbage collection.

3.4.4. DX / Workflow Metrics

- Hot Reload Speed: The time when it can be used again after the changes.
- Developer Iteration Cycle: The total time of the update, build, and reload process.
- Incremental Build Responsiveness: Especially for medium and large apps.

3.4.5. SSR & Hydration Metrics

- Server-Side Rendering Time: The way AOT/JIT impacts SSR speed.
- Hydration Performance: The merging cost of the server-rendered DOM with the client state.
- All these features together cover not only the production but also the development scenarios.

3.5. Data Collection Tools

Several different measurement instruments will be put to use for accuracy and repeatability:

- Chrome Lighthouse: Records Web Vitals and runtime metrics.
- WebPageTest: For network-oriented and real-device simulations.
- Angular CLI Bundle Analyzer: Helps to see and count the bundle parts.
- Node/CLI Timing Logs: Tracks build-time performance for AOT and JIT.
- Chrome DevTools Performance Panel: Identifies JS execution, memory, and CPU usage.
- Memory Profiler Tools: Monitors heap snapshots and memory changes during runtime.
- Browser Tracing (Performance API): Used for very detailed timeline investigation.

If you employed multiple tools, the likelihood of your results being biased or incomplete would be reduced.

3.6. Data Analysis Approach

The data recorded will be thoroughly reviewed through statistical as well as comparative methods.

- Comparative tables: AOT vs JIT metrics analysis categorized by application size and configuration.
- Visualizations: graphs illustrating construction durations, bundle dimensions, and performance trajectories
- Statistical Assessment: Calculation of mean, variance, and confidence intervals for the most relevant instances.
- Trend Identification: Finding points where AOT is performance-wise consistently better than JIT, e.g., large-scale builds or SSR workflows.
- Contextual Interpretation: Connecting the observed performance with Ivy's architectural features, such as locality and instruction generation.

The intent is to convert the raw data into insights that developers and organizations can use to make the right choice of compilation strategy under Ivy.

4. Case Study

4.1. Overview

This segment, together with the empirical benchmarks and the theoretical analysis, brings forward a detailed case study which evaluates the real-world impact of the choice between Ahead-of-Time (AOT) and Just-in-Time (JIT) compilation in an Angular Ivy–based mid-sized production application. Controlled experiments offer stable numerical comparisons, but examining different compilation modes in a dynamic, actively maintained codebase shows practical challenges, workflow patterns, and deployment issues that cannot be easily simulated. The case study revolves around an enterprise analytics dashboard that was developed with Angular Ivy and deployed in a production environment with diverse user profiles, complex data interactions, and multi-layered architecture. By means of actual implementation, the study unveils the subtle effects of AOT and JIT on development cycles, performance bottlenecks, CI/CD pipelines, SSR workflows, and user experience, particularly, for low-end mobile devices that frequently access the dashboard. The goal is to gather more insights that could help the engineering teams in deciding which compilation strategy is the most compatible with their technical and business requirements.

4.2. Application Context

The application in question is an analytics dashboard for a mid-sized enterprise internal business teams and external clients. It consists of several feature modules integration, each representing a domain-specific workflow such as reporting, forecasting, KPI monitoring, and real-time event tracking. The modules mainly implement reusable components, thus they reflect modern Angular development patterns which were introduced after Ivy. The dashboard is heavily decorated with data visualizations that are powered by charting libraries and are using custom rendering logic that is optimized for handling large datasets. It depends on both REST APIs for persistent data retrieval and WebSocket connections for live data updates, thus, real-time analytics and performance monitoring are enabled.

The architectural design focuses on modularity, lazy loading, and route-based segmentation to ensure scalability. As the user interactions are frequent and the real-time dashboards are needed, the application is required to retain smooth runtime performance and at the same time, the initial load time should be as short as possible. The user base is equipped with devices ranging from high-end desktop machines to low-end mobile devices which are commonly used by field operators. The application's diversity makes it a perfect test ground for studying the behavior of AOT and JIT in real-world scenarios that are sensitive to performance.

4.3. Evaluation Parameters

Different parameters were set to measure how AOT and JIT had an impact on the application performance and development workflow:

- Build Complexity: The modular structure of the dashboard, together with the heavy usage of visualization libraries and extensive routing, provided a varied test surface for the analysis of build times and bundle composition resulting from different compilation modes. The investigative focus was mainly on how the Ivy's locality influenced the incremental rebuild behavior.
- Deployment Pipeline Variations: The application is using a multi-stage CI/CD pipeline that includes code linting, testing, pre-production deployments, and automated bundle optimizations. The comparison between AOT and JIT has shown the differences in caching, reproducibility, artifact size, and deployment duration.
- SSR Impact Using Angular Universal: As the dashboard provides partial SSR support for SEO reasons and the improvement of the first contentful rendering, the case study has been led to analyze how the compilation mode affected SSR boot times, server CPU usage, and hydration performance on the client.
- CI/CD Integration Challenges: The multi-branch development, hotfix pipelines, and daily production deployments of the project brought in intricate issues related to build caching, dependency updates, and performance regression testing.

- The team decided to test the dashboard on artificially slowed environments and low-end Android devices to collect metrics like LCP, FID, and responsiveness, thereby ensuring the authenticity of the user experience and comparing the JIT-compiled vs AOT-compiled deployments.

4.4. Observations during Implementation

- Developer Workflow Differences
  T clearly provided faster iteration cycles during local development that were less than one minute, especially when working on components with dynamic templates or frequent UI refinements. Dev servers refreshed more quickly, and changes appeared almost instantly. In comparison, development builds with AOT took longer to rebuild and iterative UI work was slower, even though the Ivy changes had improved those times. Developers usually took JIT advantages when prototyping dashboards or experimenting with visualization layouts.
- AOT-Specific Template Errors
  One of the main causes of difficulty in the implementation was the occurrence of AOT-specific template errors that the compiler might have. Stricter checks done by Ivy lead to the uncovering of the errors in the template such as ambiguous property bindings, undefined variables in structural directives, and mistyped inputs which used to be JIT builds and are not always caught by them. While at the very beginning the disruption was noticeable, these failures eventually resulted in better template verification and the elimination of runtime errors in production.
- Lazy Loading Behavior
  Lazy-loaded modules demonstrated different behavior under AOT vs JIT conditions. Ivy generated more optimized factory code that shortened the time needed for loading additional modules after the initial route in AOT-enabled builds. Whereas under JIT modules took slightly longer to load because template parsing was done during runtime. For a dashboard with numerous on-demand screens, this difference noticeably impacted navigation performance on low-end devices.
- Cache Invalidation & Re-deployment
  CI/CD pipelines could utilize stable caching from their runs due to the consistent, deterministic output of AOT builds. JIT builds had bigger artifacts and occasionally that led to cache invalidation in an unanticipated manner because of changes in template compilation. With strict caching policies and CDN-backed static serving in deployment environments, fewer invalidations were produced by AOT and thus the overall redeployment time was dropped. Nevertheless, in a few cases, JIT was as well staged to a rapid workflow and hence testable builds were prioritized.

Table 1: Developer Experience & Workflow Trade-offs

Dimension	JIT (dev)	AOT (dev / prod)	Recommendation
Iteration speed	Best — fastest hot reload	Slower initial rebuilds (but improved by Ivy locality)	Use JIT during rapid prototyping
Template error detection	Less strict at dev runtime	Stricter; catches more issues early	Consider AOT for CI/pre-prod tests
Cache stability	Larger artifacts → more cache churn	Deterministic outputs → fewer cache invalidations	Use AOT in CI/CD and CDNs
Rebuild responsiveness	Fast	Fast in incremental mode (Ivy)	Consider hybrid: JIT dev + AOT staging/prod

4.5. Summary of Case Study Insights

AOT is the major factor in significantly improving the initial load performance as per the case study, particularly for users of low-end mobile devices where CPU and network constraints can double the benefits of precompiled and tree-shaken output. On the contrary, JIT is still needed for fast prototyping, experimental UI, and local development workflows where build speed is more of a runtime inefficiency. Ivy makes the difference between AOT and JIT smaller in such aspects as build speed and output size, but the distinctions are still quite noticeable in the real world for SSR, heavy visualization, and complex routing use cases. In general, these findings confirm that both compilation modes can be used, production environments greatly benefit from AOT, and JIT still has development cycles.

5. Results and Discussion

5.1. Summary of Benchmark Results

The benchmark comparison of small, medium, and large Angular Ivy applications reveals that the patterns of AOT and JIT performance are quite similar in all three types of apps. The experimental findings show that AOT creates bundles that are much smaller, with the size of the bundles being cut by 20% for small apps and by almost 40% for large-scale applications. The main reason for this reduction is that Ivy employs very aggressive tree shaking and instruction-set-based template compilation, thus production builds no longer have to ship template metadata and compiler functions.

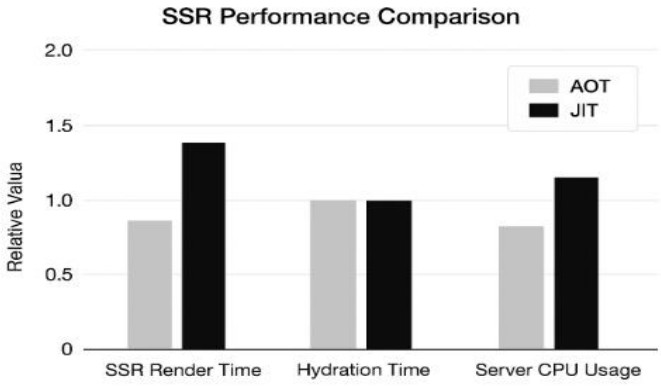

Fig 2: SSR Performance Comparison

Concerning build-time metrics, the situation is more complicated. JIT compilation is still quicker for initial builds, thus local dev servers or isolated builds without caching should be preferred if one is aiming at the speed of the first build. However, the locality model of Ivy every component is compiled separately—greatly enhances AOT rebuilds. In medium and large applications, incremental AOT builds can sometimes be faster than JIT simply because Ivy only recompiles those components that have been changed and does not traverse the whole dependency graph.

Runtime performance outcomes are just as clear. AOT builds lead to less CPU usage, less memory consumption, and shorter time-to-interactive (TTI), especially on heavily loaded mobile devices. JIT builds, on the other hand, perform template parsing and code generation in the browser, thus they have extra runtime costs. The differences become even larger in the big enterprise app where the CPU overhead of JIT is the main reason for the drop of frames and the delay of user interaction.

SSR performance is another factor that strongly supports AOT. As AOT precompiles templates into fixed instructions, SSR render times are therefore 10–25% faster, and hydration is carried out more efficiently because of less client-side parsing. On the other hand, JIT, which has compiler overhead in its runtime, leads to SSR bootstrapping being slower and server CPU usage being higher.

The differences between AOT and JIT that still exist in bundle size, runtime performance, and SSR behavior have been confirmed by the results. Ivy has closed the historical gaps between AOT and JIT, which are the two compilation strategies, to some extent.

5.2. Interpretation of Performance Metrics

5.2.1. Build-Time Trade-offs

By localities-based compilation Ivy has dramatically cut the overhead of AOT builds that used to be very high in the View Engine era. Nevertheless, JIT is still unbeaten in very quick rebuild cycles which is why this is the preferred way of working for developers who are actively working on the code and UI. Incremental builds with Ivy make AOT much more powerful, especially when cache layers (for instance Angular CLI persistent build cache or Nx computation caching) are activated. In giant apps, the difference in speed between AOT and JIT for rebuilding is so small that it shows Ivy has done a lot to increase the developer's workflow even when working under AOT.

5.2.2. Bundle Size Considerations

AOT can produce smaller bundles than JIT because the templates are precompiled into very concise instruction sets. There is no need therefore to include Angular’s template compiler or large metadata objects that are necessary for JIT parsing in the bundle. In JIT bundles, template definitions and compilation utilities have to be there and therefore payloads of bundles will be heavier leading to longer network transfer time. On top of that, Ivy’s better tree shaking technology widens the gap more since whichever instructions and features that are not used in the case of AOT bundles can be completely removed, while JIT bundles will often retain the broader compiler logic even if it is not used at all. So, for production environments where the first priority is mobile or the bandwidth is low, the reductions in the bundle size that AOT offers can be translated into real performance benefits.

5.2.3. Runtime Performance

The runtime metrics greatly support the use of AOT. This is because runtime template parsing, initial rendering, change detection setup, and component instantiation are all done faster in AOT builds. JIT, on the other hand, is said to be a source of CPU work at runtime, which is unavoidable; hence, it causes slow startup times and increased memory usage—especially on mid-range and low-end devices where JavaScript execution speed is limited. With Ivy’s optimized template instructions, the

JIT overhead is reduced when compared to the earlier Angular versions, but this does not mean that the cost of transforming template instructions during runtime is completely done away. Production performance in AOT is, therefore, more stable and predictable.

5.2.4. Developer Experience

Developer experience (DX) is still an area where JIT outperforms, especially in the case of iterative UI development. JIT is perfect for development environments due to faster hot reload, lower initial build overhead, and minimal template experimentation friction. Nevertheless, AOT also helps DX by making it more obvious and rigorous in terms of detecting template errors. Many of the structural or binding issues that are JIT tolerant may cause a failure under AOT, thus, production bugs that are subtle are prevented. A team using AOT for development may require faster hardware or caching layers, but the advantage is in early error detection and build accuracy improvement.

5.2.5. SSR & SEO Considerations

SSR processes become more efficient with AOT. When templates are precompiled, the servers can render the pages not only quicker but also more stable, with a lower CPU usage and a reduced memory footprint. It is a great advantage to an enterprise SSR deployment with high traffic as server efficiency directly translates into operational cost. The reason for this is that client-side hydration can also be fast with AOT as the templates have already been converted into Ivy instructions. The use of JIT at runtime leads to on-the-fly compilation overhead that causes server slowdowns as well as time-to-first-byte (TTFB) increase, thereby, SEO-critical metrics get affected.

5.2.6. Trade-offs Summary

Table 2: Comparison between AOT (Ahead-of-Time) and JIT (Just-in-Time) compilation techniques

Feature	AOT	JIT
Bundle Size	↓ Smaller	↑ Larger
Runtime Speed	↑ Fast	↓ Slower
Build Time	Often slower initially	Fastest
Rebuild Speed	Fast (Ivy locality)	Fast
Developer Experience	Moderate	Best
SSR Performance	Best	Moderate

5.3. Implications for Production

Such results are very clear: AOT should be the strategy of choice for production deployments, in particular, performance-sensitive, SEO-dependent, or mobile-targeted applications. The wins in bundle size, runtime efficiency, SSR performance, and deterministic rendering translate directly into getting more real-world user and server infrastructure minutes.

Nevertheless, a few scenarios which use JIT still be conceptually valid:

- Fast UI prototyping
- Initial development stages with frequent template changes
- Sandbox or experimental environments
- Cases where ultra-fast rebuild loops without AOT overhead are necessary

The enhancements made by Ivy facilitate the switch between the two modes, but the fundamental compromises still exist. In the end, the decision of teams should result in a hybrid workflow: JIT during early development and AOT for staging, performance testing, and production deployment.

6. Conclusion and Future Scope

6.1. Conclusion

The head-to-head performance comparison of Ahead-of-Time (AOT) and Just-in-Time (JIT) compilation in Angular's Ivy engine reveals that Ivy significantly reduces the difference in performance between the two compilation methods but still does not completely eliminate the difference. Both local-based compilation of Ivy, instruction-based rendering pipeline, and enhanced tree-shaking capabilities make the two modes more efficient and predictable. However, the findings still show that AOT is still the most powerful choice for production environments, as it can produce smaller bundles, faster runtime, better SSR performance, and more efficient hydration. These benefits lead directly to improved user experiences, especially in mobile-first and low-bandwidth scenarios.

On the other hand, JIT is still very important in the modern Angular development processes as it provides the fastest speed for the UI-building iteration, rapid prototyping, and interactive debugging. Its versatility and low initial build overhead keep JIT as the developers' favorite in active development situations although it has some limitations at runtime.

From real-world benchmarks and practical experiments, a hybrid development pipeline, which involves using JIT in the early stages of development and then switching to AOT for staging, testing, and production, is still the most efficient and popular workflow. Ivy supports this hybrid pattern by lessening the AOT build inconvenience and making incremental build behavior better, thus allowing teams to have both quick developer iteration and great user performance.

6.2. Future Scope

Looking forward, a handful of newfangled ideas will continue to influence Angular's compilation workflows. As Angular merges Esbuild, Vite, and future Rust-bundlers based tools like to its environment, universal AOT-based development pipelines of a faster kind may even be possible with the JIT dependency being reduced or completely eliminated in development. Partial hydration, Ivy-native SSR refinements, and standalone components maturity being some of the next trends for Angular that would probably have an impact on compilation strategies as well as performance characteristics.

Moreover, the presence of a substantial amount of potential that can be unlocked to the great extent of exploring incremental AOT-only development modes which make it possible to have rebuilds at speeds that are almost at the level of JIT and at the same time production-grade compilation benefits is also left open. On top of that, AI-powered compiler refinements like automated template analysis, error prediction, and pre-processing could potentially revamp the future of compilation in terms of performance and developer experience.

References

[1] Smith, Todd, et al. "Practical experiences with Java compilation." *International Conference on High-Performance Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

[2] Rohou, Erven. *Infrastructures and Compilation Strategies for the Performance of Computing Systems*. Diss. Université de Rennes 1, 2015.

[3] Ronaghi, Zahra, et al. "Python in the NERSC exascale science applications program for data." *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*. 2017.

[4] Parakala, Adityamallikarjunkumar. "Citizen-Facing Automation: Chatbots and Self-Service in Public Services." *International Journal of AI, BigData, Computational and Management Studies* 4.4 (2023): 108-118.

[5] Marx-Raacz Von Hidvég, Tomas. "Are the frameworks good enough? A study of performance implications of JavaScript framework choice through load-and stress-testing Angular, Vue, React and Svelte." (2022).

[6] Connolly, Gladys Twining. "A Cost-Performance Analysis of Computer Alternatives." (1982).

[7] Pan, Alexander, et al. "Do the rewards justify the means? Measuring trade-offs between rewards and ethical behavior in the machiavelli benchmark." *International conference on machine learning*. PMLR, 2023.

[8] Duffie, Darrell, Piotr Dworczak, and Haoxiang Zhu. "Benchmarks in search markets." *The Journal of Finance* 72.5 (2017): 1983-2044.

[9] Xie, Saining, et al. "Rethinking spatiotemporal feature learning: Speed-accuracy trade-offs in video classification." *Proceedings of the European conference on computer vision (ECCV)*. 2018.

[10] Coleman, Cody, et al. "Dawnbench: An end-to-end deep learning benchmark and competition." *Training* 100.101 (2017): 102.

[11] Batchu, Krishna Chaitanya. "Modern Data Warehousing in the Cloud: Evaluating Performance and Cost Trade-offs in Hybrid Architectures." *International Journal of Advanced Research in Computer Science & Technology (IJARCST)* 5.6 (2022): 7343-7349.

[12] Parakala, Adityamallikarjunkumar. "Vendor Highlights–IoT, AI, and Process Mining." *International Journal of Emerging Trends in Computer Science and Information Technology* 4.4 (2023): 135-146.

[13] Nugteren, Cedric, et al. "High performance predictable histogramming on gpus: exploring and evaluating algorithm trade-offs." *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. 2011.

[14] Sidiroglou-Douskos, Stelios, et al. "Managing performance vs. accuracy trade-offs with loop perforation." *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011.

[15] Guntupalli, Bhavitha. "Data Lake vs. Data Warehouse: Choosing the Right Architecture." *International Journal of Artificial Intelligence, Data Science, and Machine Learning* 4.4 (2023): 54-64.

[16] Webb, Nicholas P., et al. "Indicators and benchmarks for wind erosion monitoring, assessment and management." *Ecological Indicators* 110 (2020): 105881.

[17] Rachuri, Kiran K., et al. "Sociablesense: exploring the trade-offs of adaptive sampling and computation offloading for social sensing." *Proceedings of the 17th annual international conference on Mobile computing and networking*. 2011.

[18] Ciric, Rastko, et al. "Benchmarking of participant-level confound regression strategies for the control of motion artifact in studies of functional connectivity." *Neuroimage* 154 (2017): 174-187.

[19] Vamshidhar Reddy Vemula.(2023).Multi-Cloud Security Orchestration Using Deep Reinforcement Learning.