ChatGPT

# Proje İçin Tam Yığın Teknoloji Seçimi ve Mimari Önerisi

## Frontend Framework Karşılaştırması (Next.js, SvelteKit, Nuxt.js, Astro)

**Next.js (React tabanlı):** Next.js, React ekosisteminin gücünü alarak tam yığın geliştirmeyi kolaylaştıran popüler bir framework'tür. Dahili olarak **SSR (Server-Side Rendering)**, **SSG (Static Site Generation)** ve **ISR (Incremental Static Regeneration)** desteği sunar [1] [2]. Bu sayede hem dinamik içerikli sayfaları sunucuda işleyip hızlı yükleme ve SEO avantajı elde edebilir, hem de statik sayfaları önceden üretip CDN üzerinden sunabilirsiniz. Next.js ayrıca **güçlü bir görüntü optimizasyonu** ve **kod bölme** mekanizmasına sahiptir; büyük ölçekli uygulamalarda bile performansı korumaya yardımcı olur [3] [4]. Ancak React tabanlı olması nedeniyle başlangıç paket boyutu Svelte gibi derleyici tabanlı yaklaşımlara göre daha büyük olabilir [5]. Geliştirici deneyimi açısından Next.js olgun bir ekosisteme sahiptir – 100 binden fazla GitHub yıldızı ve kapsamlı dokümantasyonuyla geniş bir topluluk desteği bulunur [6]. React bilen ekipler için Next.js öğrenmesi kolaydır; ancak React'e aşina olmayanlar için JSX ve ekosistem öğrenme eğrisi oluşturabilir [7].

**SvelteKit (Svelte tabanlı):** SvelteKit, sanal DOM kullanmayan **derleyici tabanlı** Svelte kütüphanesinin meta-framework'üdür. En büyük avantajı, **ön uç kodunu derleme anında optimize etmesi** ve neredeyse sıfır çalışma zamanı eklemesidir. Bu nedenle SvelteKit uygulamalarının başlangıç yüklenme süresi çok hızlıdır – bazı ölçümlere göre SvelteKit, React tabanlı framework'lere kıyasla sayfaları ~**%50 daha hızlı** yükleyebilmektedir [8]. SvelteKit de SSR, SSG ve ISR'ı destekler, böylece Next.js'e benzer esneklik sunar [9] [10]. Oluşturduğu paketler küçük olduğu için ilk yanıt süresi ve etkileşim süresi mükemmeldir [11] [12]. Ayrıca SvelteKit, Svelte'nin reaktif durum yönetimi sayesinde harici bir state kütüphanesine ihtiyaç duymaz; **dahili store mekaniği** ile basit ve anlaşılır bir geliştirme deneyimi sağlar [13] [14]. Geliştirici deneyimi yönüyle, SvelteKit "az kod ile çok iş" felsefesini benimser – Svelte sentaksı daha anlaşılır ve bileşen yapısı daha minimalisttir, bu da yeni başlayanlar için bile hızlı bir başlangıç sunar [15]. Ancak SvelteKit ekosistemi React/Next kadar büyük değildir (GitHub'da ~17 bin yıldız) ve topluluk daha küçüktür [16]. Yine de hız ve basitlik öncelikli projelerde SvelteKit oldukça cazip bir seçenektir [17] [18].

**Nuxt.js (Vue 3 tabanlı):** Nuxt, Vue.js dünyasının Next.js karşılığıdır. Vue 3 tabanlı olduğundan **öğrenmesi görece kolay** (React'e göre) ve Vue bilen ekipler için doğal bir seçimdir [19]. Nuxt 3, **Nitro** adlı yeni motoruyla Node.js dışında serverless platformlar ve edge (kenar) ortamlarda da çalışabilen esnek bir SSR altyapısı sunar [20]. Dosya tabanlı rotalama, otomatik bileşen içe aktarma gibi özelliklerle Vue projelerinde sıfırdan yapılandırma ihtiyacını ortadan kaldırır [21] [22]. **Render esnekliği:** Nuxt, sayfa bazında CSR/SSR/SSG tercihi yapmanıza izin verir; örneğin bazı sayfaları statik, bazılarını SSR olarak sunabilirsiniz [23]. Bu sayede pazarlama sayfaları için statik hız ve SEO, dashboard gibi sayfalar için SSR ile canlı veri avantajı aynı projede mümkün olur. Nuxt'nin **modül** sistemi, kimlik doğrulama, analiz, PWA gibi sık kullanılan özellikleri kolayca entegre etmeyi sağlar [24]. Performans olarak Vue 3 önemli iyileştirmeler getirse de, SvelteKit kadar hafif değildir; ancak orta ve büyük ölçekli uygulamalarda Nuxt'nin otomatik kod bölme, önbellekleme ve optimizasyonları sayesinde yeterli performans elde edilir. Geliştirici deneyimi ve üretkenlik açısından Nuxt son derece olumludur – tek komutla proje başlatma, sıcak modül yenileme, tür desteği, vs. kutudan çıktığı için altyapıyla uğraşma süresini en aza indirir [22]

[25]. Vue ekosistemi React kadar büyük olmasa da hızla büyümektedir; Nuxt özelinde de geniş bir eklenti/modül havuzu oluşmuştur. Vue tercih eden ekipler için Nuxt, **düşük öğrenme eğrisiyle** tam yığın geliştirme imkanı verir.

**Astro:** Astro, diğerlerinden biraz farklı bir yaklaşıma sahip yeni nesil bir framework'tür. Astro'nun odağı, içerik ağırlıklı sitelerde **mümkün olan en az JavaScript ile en yüksek performansı** sunmaktır. Varsayılan olarak sayfaları derleme zamanında statik HTML olarak üretir (SSG) ve istemciye **"0 KB JavaScript"** ile gönderir; sadece etkileşim gereken bileşenler için **ada mimarisi** kullanarak gerektiğinde kısmi JS yükler [26] [27]. Bu yaklaşım sayesinde Astro siteleri inanılmaz hızlıdır – yapılan bir karşılaştırmada Astro ile oluşturulan bir site, popüler React tabanlı bir siteye kıyasla ~**%40 daha hızlı yüklendi** ve %90 daha az JS gönderdi [26]. Bu da Core Web Vitals ve kullanıcı deneyimi metriklerinde büyük avantaj demektir [28] [29]. Astro **çoklu framework desteği** sunar; React, Svelte, Vue bileşenlerini aynı projede adalar olarak kullanabilirsiniz [30] [31]. SEO bakımından, Astro tüm içeriği önceden HTML olarak sunduğu için arama motorları sayfaları kolayca tarar – SSR veya SSG ile üretilen tam HTML sunulduğundan SEO açısından mükemmeldir [32] [33]. Ancak Astro, tam etkileşimli web uygulamalarından ziyade blog, doküman, pazarlama sitesi gibi statik ağırlıklı projeler için idealdir [34] [35]. Sık değişen veya anlık kullanıcı etkileşimi gereken uygulamalarda Next.js/Nuxt gibi SSR çözümleri daha uygun olabilir, çünkü Astro'nun SSR desteği sınırlıdır (yalnızca gereken yerlerde, isteğe bağlı SSR yapabilir) [2]. Geliştirici deneyimi olarak Astro sade bir yapıya sahiptir; içerik odaklı geliştiriciler Markdown/MDX desteğiyle yazılarını kolayca entegre edebilir [36]. Topluluğu henüz küçük fakat aktif ve astro/add komutlarıyla yapılandırma zahmetini en aza indiren modern bir ekosistemi vardır [37] [38]. Özetle, tamamen içerik odaklı ve performansın kritik olduğu durumlarda Astro öne çıkar, ancak interaktif SaaS uygulamalarında Next/SvelteKit gibi genel amaçlı framework'ler daha esnek olacaktır [39] [40].

**Performans, SEO, Streaming ve DX Özet:** Next.js, SvelteKit, Nuxt ve Astro'nun tümü SSR desteği sayesinde SEO dostudur – sayfalar sunucuda işlendiği için arama motorları tam içerik görebilir [41] [42]. Next.js ve Nuxt, **bütüncül tam yığın çözümler** olarak hem istemci hem sunucu tarafında zengin özellikler sunar; büyük toplulukları sayesinde geliştirme sırasında karşılaşılan sorunlara çözüm bulmak kolaydır. SvelteKit, **performans kritiktir** diyen projeler için küçük paket boyutu ve hızlı yükleme avantajıyla çekicidir; geliştiriciye de daha **basit bir kod tabanı** sunar [43] [44]. Astro ise içerik sitelerinde **en iyi Core Web Vitals** skorlarını elde etmek için idealdir – gönderilen JS miktarını en aza indirerek kullanıcıya odağı içeriğe verir. *Streaming desteği* konusuna gelirsek: Next.js, React 18 ile birlikte **streaming SSR** (parçalı HTML akışı) yeteneğini de deneyselleştirmektedir, bu sayede büyük sayfaların parça parça kullanıcıya aktarılması mümkün olur [45]. SvelteKit de Node tabanlı çalıştığından benzer şekilde akış yanıtlar gönderebilir; ayrıca tüm framework'ler, tarayıcı tarafında **SSE (Server-Sent Events)** veya WebSocket ile gelen akışları işleyebilir durumdadır. Örneğin, bir arka uçtan parça parça gelen yanıtları EventSource veya fetch stream API ile alıp kullanıcıya göstermek, framework fark etmeksizin mümkündür. Geliştirici deneyiminde Next.js/Nuxt gibi olgun framework'ler daha **bol kaynak ve araç desteğine** sahipken, SvelteKit/Astro gibi yeniler **daha az konfigürasyon** ile işe koyulma avantajı sunar. Ekip yetkinliği de seçimde önemlidir: Halihazırda React bilgisine sahip bir ekip için Next.js ile üretkenlik yüksek olacaktır; Vue bilenler Nuxt ile rahat eder; yeni bir yaklaşım denemek isteyen ve maksimum performans isteyenler SvelteKit'i seçebilir. Sonuçta, **projenin öncelikleri** (performans mı, hızla geliştirme mi, SEO mu, topluluk desteği mi?) hangi frontend teknolojisinin en uygun olduğunu belirleyecektir [46].

*Özet bir tabloyla karşılaştırma:*

| Framework | Temel Teknoloji | Render Yöntemleri | Performans & Yükleme Süresi | SEO & Özellikler | Geliştirici Deneyimi (DX) |
|---|---|---|---|---|---|
| **Next.js** | React (JS/TS) | SSR, SSG, ISR, CSR | Yüksek ancak React kaynaklı **daha büyük bundle** olabilir; kod bölme + optimizasyon ile telafi edilebilir [47] . | Mükemmel SEO (SSR+SSG) ve meta yönetimi [41] ; **Resim optimizasyonu, API route** desteği. | Esnek ama konfigürasyon gerektirebilir; React bilenler için rahat, büyük topluluk (119k+ star) [6] . |
| **SvelteKit** | Svelte (JS/TS) | SSR, SSG, ISR | **Çok hızlı** başlangıç yüklemesi (önceden derlenmiş, ufak JS); React tabanlı çerçevelerden ~%50 daha hızlı sayfa yükleme raporları [8] . | SSR/SSG sayesinde SEO dostu; otomatik resim ve kod optimizasyonu [9] . | **Minimal konfigürasyon, dahili state** ile basit kullanım; küçük ama hızla büyüyen topluluk (17k star) [16] . |
| **Nuxt 3** | Vue 3 (JS/TS) | SSR, SSG, ISR, CSR | Vue 3 ile önceki nesle göre daha iyi performans; paket boyutu orta düzey (Svelte'den büyük, React'ten küçük). Nitro motoru ile çoklu ortam optimizasyon. | SSR/SSG ile SEO kuvvetli; **modüler** yapı (auth, PWA eklentileri), dosya tabanlı rotalama, görüntü optimizasyonu mevcut. | **Vue rahatlığı** + tam yığın özellikler; düşük öğrenme eğrisi, otomatik import ve yapılandırma [48] [22] ; topluluk ve eklenti ekosistemi büyüyor. |

| Framework | Temel Teknoloji | Render Yöntemleri | Performans & Yükleme Süresi | SEO & Özellikler | Geliştirici Deneyimi (DX) |
|---|---|---|---|---|---|
| **Astro** | Çeşitli (Island) | SSG (SSRand opsiyonel) | **Aşırı hızlı** içerik sunumu (0 KB JS başlangıç); Astro siteleri benzer Next.js sitelerinden %40 daha hızlı, %90 az JS ile yüklenebiliyor [26]. | Statik HTML çıktısı sayesinde mükemmel SEO ve Core Web Vitals [28]; sadece gereken bileşenlere JS yüklendiği için kullanıcı deneyimi akıcı. | İçerik odaklı sade geliştirme; Markdown/MDX desteği. Tam uygulamalar için kısıtlı olabilir (SSR sınırlı) [2]. Küçük ama aktif topluluk; farklı framework bileşenlerini bir arada kullanabilme esnekliği. |

## Backend Framework Karşılaştırması (FastAPI, Node.js/NestJS, Go/Fiber)

**FastAPI (Python):** FastAPI, Python 3.7+ üzerinde çalışan modern ve yüksek performanslı bir web framework'üdür [49]. Asenkron destekli Starlette altyapısı ve Pydantic ile veri modellemeyi birleştirerek, **hız ve geliştirici dostu** özellikleri dengeler [50]. Python'un avantajı olarak zengin **NLP/AI kütüphaneleri** ile doğal uyum içindedir – PyTorch, TensorFlow, Hugging Face gibi araçları doğrudan entegre etmek kolaydır. Bu nedenle makine öğrenimi veya veri işleme ağırlıklı API'ler için ideal bir seçimdir [51]. Performans açısından bakıldığında, FastAPI **Python dünyasında en hızlılarından** biri olup Uvicorn ASGI sunucusu ile birlikte kullanıldığında Node.js veya Go gibi ortamlara yakın bir performans sergilemeyi hedefler [49]. Bağımsız yapılan bazı testler, I/O ağırlıklı işlerde FastAPI'nin **binlerce isteği eş zamanlı kaldırabildiğini** göstermektedir. Yine de Python dili ve **GIL** kısıtı nedeniyle, ham işlem gücü ve eşzamanlılık konusunda Node.js ve Go'nun gerisinde kalır. Örneğin, basit bir API benchmarğinde Go ve Node.js ~80k istek/s işlemlerken, FastAPI yaklaşık 12k istek/s'de kalmıştır (aynı donanımda) [52]. Yük altında ölçeklenebilirlik için çözüm olarak FastAPI, **çalışan süreçleri arttırarak** (ör. Uvicorn/Gunicorn ile birden çok worker kullanımı) ölçeklenebilir. Ayrıca göreceli olarak daha yüksek bellek tüketme pahasına CPU yoğun işlemleri sıraya almak veya ayrı proseslere dağıtmak gerekebilir. Geliştirici deneyimi tarafında FastAPI çok başarılıdır: **Otomatik etkileşimli dokümantasyon (Swagger, ReDoc)**, tip güvenliği, Pydantic ile şema validasyonu ve basit dekoratör tabanlı API tanımı geliştirmenin hızlı ilerlemesini sağlar [50]. Topluluğu hızla büyümüş ve Python geliştiricileri arasında popüler hale gelmiştir. Özetle, eğer projeniz **doğal dil işleme, veri analizi** gibi Python ekosisteminin gücünden faydalanıyorsa ve çok aşırı trafik beklemiyorsanız FastAPI hem yeterince hızlı hem de çok esnek bir çözümdür. Hatta bazı kıyaslamalarda, I/O-bound (girdi/çıktı ağırlıklı) işlemlerde **performansının Node.js'ye yakın olduğu** belirtilmektedir [49]. CPU-bound (yoğun işlemci kullanan) görevlerde ise Python, C++ ile yazılmış kütüphaneleri kullanmadıkça daha yavaş olacaktır.

**Node.js / NestJS (JavaScript/TypeScript):** Node.js, Chrome'un V8 motoru üzerinde çalışan, tek iş parçacıklı event-loop modelini kullanan bir çalışma zamanı ortamıdır. Özellikle **I/O odaklı ve yüksek eşzamanlı** bağlantı gerektiren uygulamalarda çok başarılıdır – **non-blocking** yapısı sayesinde binlerce isteği aynı anda düşük gecikme ile işleyebilir [53] [54]. Saf Node.js üzerinde Express, Koa gibi minimal framework'ler kullanılabildiği gibi, **NestJS** gibi tam teşekküllü bir framework de tercih edilebilir. NestJS, Node.js üzerinde yapılandırılmış, Angular'dan esinlenen modüler yapısı ve dekoratörlerle **"kurumsal**

**seviye”** uygulama geliştirmeyi kolaylaştıran bir çatıdır. TypeScript desteği, iç içe modül yapısı, servisler ve bağımlılık enjeksiyonu gibi özelliklerle büyük ekiplerin Node.js ile düzenli kod yazmasını sağlar. Node.js performansına bakıldığında, tek çekirdek üzerinde çalışsa da oldukça yüksek throughput elde edilir. Örneğin bir testte Node.js tabanlı bir servis ~**80.000 istek/saniye** gibi değerler yakalayarak Python FastAPI'yi fersah fersah geride bırakmıştır [52]. Node'un ham performansı C++ derleyici optimizasyonu (V8) sayesinde yüksektir [55]. Ancak Node'un da sınırları vardır: Tek iş parçacığı olduğu için aynı anda CPU'ya yüklenen ağır işlemler (ör. büyük ML modeli çalıştırmak) tüm akışı bloklayabilir. Bunu çözmek için işçi iş parçacıkları (Worker Threads) veya ayrı mikroservisler gerekir. NestJS özelinde, Node'un performans avantajını korurken geliştirici deneyimini artırır; yapısal bir yaklaşımla çok sayıda kütüphane ve hazır modül barındırır. Örneğin kimlik doğrulama, valideasyon, logging gibi konularda NestJS zengin bir eklenti ekosistemine sahip. Node/Nest'in en büyük avantajı, **tek dil ile tam yığın** geliştirme imkanı vermesidir – frontendi React/Next ile yazan bir ekip backend'i de TypeScript ile yazarak dil birliğini koruyabilir. Ayrıca Node ekosisteminde NPM üzerinden hemen her ihtiyaç için bir paket bulunur. Dezavantaj olarak, doğası gereği Python kadar AI/ML odaklı değildir; eğer NLP modellerini kullanmak isterse ya Python hizmetlerine istek göndermek ya da TensorFlow.js gibi kısıtlı JS kütüphaneleriyle yetinmek durumunda kalabilir. *Verimlilik* açısından Node oldukça iyi ölçeklenir: Tek bir süreç birden çok çekirdeği kullanamasa da PM2 gibi işlem yöneticileriyle veya container ölçeklendirmesiyle yatay büyüme sağlanabilir. Hafıza kullanımı Go gibi dillere nazaran biraz yüksek olabilir ancak genelde web sunucusu olarak kabul edilebilir seviyededir. Yüksek eşzamanlı sohbet, anlık bildirim gibi görevlerde Node.js'nin performansı kanıtlanmıştır. NestJS'in getirisi olan yapılandırma ise başlangıçta öğrenme eforu gerektirebilir; framework sözleşmelerine ve TypeScript'e uyum sağlamak gerekir. Özetle, **çok yüksek trafik ve gerçek zamanlılık** gereken, ve ekipte JavaScript uzmanlığı bulunan durumlarda Node/NestJS mükemmel bir tercihtir. Ancak saf performans kritikliğinde Go gibi dillere yetişemeyebilir (örneğin Go Fiber, Node Express'ten ~7.5 kat hızlı ölçülmüştür) [56].

**Go / Fiber (Golang):** Go, statik tipli derlenen bir dil olarak, web hizmetlerinde **az kaynakla maksimum performans** hedefleyen ekipler için öne çıkar. Go'nun standart kütüphanesi dahi oldukça hızlı bir HTTP sunucusu içerir. Fiber gibi framework'ler ise Go'nun performansını daha da kolay kullanılabilir hale getirir. Fiber, Go'nın **fasthttp** kütüphanesini temel alarak Express.js benzeri bir API sunar, ancak alttaki Go gücü sayesinde son derece hafiftir. Go'nun **goroutine** modeli ve hafif iş parçacıkları, aynı anda çok sayıda isteği paralel olarak işlemeye olanak tanır; bu concurrency modeli sayesine Go uygulamaları genellikle CPU çekirdeklerini tam kapasiteyle kullanarak müthiş throughput elde eder. Gerçek dünyadan bir karşılaştırma: Basit bir REST servisini test eden bir deneyde Go Fiber, 2 CPU'lu bir makinede ~**85.000 istek/saniye** gibi bir değere ulaşıp Node.js'i hafifçe geride bırakmıştır (aynı senaryoda Node ~80k, Python ~12k civarı) [52]. Dahası, veritabanı ve önbellek gibi bileşenlerle gerçekçi yük altında bile Go ~15k istek/s civarında stabil kalırken, Python FastAPI ~3.600'e düşebilmektedir [57]. Yani ölçeklenebilirlik ve ham hız anlamında Go zirvededir. Fiber veya Gin gibi framework'ler minimal soyutlamalarla bu hızı korur. **Verimlilik (efficiency)** açısından da Go dikkat çekicidir: Derlenmiş kod ve çöp toplayıcı kombinasyonu, bellek ve CPU kullanımında oldukça tasarrufludur. Bu nedenle yüksek trafikli sistemlerde genellikle daha az sunucuyla aynı yükü taşıyabilir, uzun vadede altyapı maliyetlerini düşürebilir [58] [59]. Elbette Go'nun da dezavantajları var: Python veya Node ekosistemindeki kadar zengin bir NLP/AI kütüphanesi yoktur. Eğer proje metin işleme, PDF okuma, özetleme gibi işler yapacaksa, Go ile ya harici servisler çağırmak (ör. bir Python mikroservisi) ya da mevcut C/C++ kütüphanelerini Go paketleriyle sarmalamak gerekebilir. Geliştirici deneyimi yönünden Go, basit ve sade sözdizimiyle övgü alsa da dinamik dillere alışkın ekiplerde başlangıçta üretkenlik kaybı olabilir. Fiber gibi framework'ler Express benzeri olduğu için web API geliştirme öğrenme süresini azaltır. Ayrıca Go, güçlü tip sistemi sayesinde derleme aşamasında pek çok hatayı yakalar, bu da büyük kod tabanlarında güven verir. Özetle, **en yüksek performansın kritikliği** ve minimum gecikmenin şart olduğu, aynı zamanda geliştirici ekibin Go konusunda bilgili veya öğrenmeye istekli olduğu durumlarda Go/Fiber seçeneği rakipsiz hız sunacaktır. Örneğin bir benchmark sonucuna göre, çok çekirdekli senaryoda Go Fiber, Express.js'ten ~**7.5 kat**, FastAPI'den ~**11 kat** daha hızlıdır [56]. Ancak bu kazanımlar, ancak sistemin gerçekten yüksek yük

altında olduğu durumlarda anlamlı olacaktır. Düşük-orta trafiğe sahip bir uygulamada Python veya Node'un performansı genelde yeterlidir ve geliştirme hızındaki konfor daha önemli hale gelir. Bu nedenle, Go tercih ederken ekibin buna hazır olması ve projenin ölçeğinin bunu gerektirmesi beklenir.

*Backend performans özet tablosu:*

| Framework | Dil (Platform) | Eşzamanlılık Modeli | İşlem Hızı (Ham Benchmarks) | Güçlü Yanlar | Zayıf Yanlar / Uyarılar |
|---|---|---|---|---|---|
| **FastAPI** | Python 3 (Uvicorn) | Async IO (çoklu worker desteği) | ~12k istek/sn (2 CPU, ham API) [52] (tek süreç); çoklu süreçle orantılı artabilir. | Python ekosistemi (NLP/ML) ile doğal uyum; **otomatik şema & dokümantasyon**; geliştirici dostu. **Yüksek I/O verimi** (async) [50] . | Tek çekirdek performansı Node/Go'dan zayıf; **CPU-bound** işler GIL nedeniyle kısıtlı (arka plana taşınmalı). Çok trafik için daha fazla ölçeklendirme gerekir. |
| **NestJS (Node)** | JS/TS (Node V8) | Event Loop (tek thread, async I/O) | ~80k istek/sn (2 CPU, cluster ile çekirdek başına ~40k) [52] . Yüksek eşzamanlı bağlantı verimi. | **Tek dilde tam yığın** (JS/TS); muazzam paket ekosistemi; event-driven yapı ile **düşük gecikme** [55] . NestJS ile yapısal ve ölçeklenebilir kod, geniş topluluk. | **CPU yoğun işler** için uygun değil (tek thread tıkanır); çözüm olarak işçi thread veya ayrı servis gerekir. Bellek kullanımı Go'ya kıyasla biraz yüksek olabilir. NLP/ML için doğrudan destek sınırlı, entegrasyon gerekir. |

| Framework | Dil (Platform) | Eşzamanlılık Modeli | İşlem Hızı (Ham Benchmarks) | Güçlü Yanlar | Zayıf Yanlar / Uyarılar |
|---|---|---|---|---|---|
| **Go Fiber** | Go (derlenen) | Goroutine (çoklu thread, M:N) | ~85k istek/sn (2 CPU, tam paralel) [52]. Fiber ile <1 sn'de 5000 istek işlenebilir (Express'ten 7-8x hızlı) [60] [61]. | **Sınıfının en hızlısı**: Düşük gecikme, düşük kaynak kullanımı; yüksek **çoklu işlem** performansı. Derlenen binary, kolay dağıtım; goroutine ile bloklamayan yüksek paralellik. | Kütüphane ekosistemi web dışı alanlarda nispeten küçük (özellikle NLP alanında). Geliştiricilerin Go diline hakimiyeti yoksa öğrenme eğrisi olabilir. Dinamik ihtiyaçlar için her şey kodlanmalı (daha az sihirli kolaylık). |

## Python Backend ile Frontend Entegrasyonu (PDF Yükleme, NLP Sonuç İletimi, Streaming, Asenkron Görevler)

Projemizde öngörülen mimaride frontend (Next.js gibi) ile backend (FastAPI gibi Python) ayrı katmanlar olarak çalışacaktır. Bu iki katmanın **entegrasyonu** kritik konular şunlardır:

- **PDF Dosya Yükleme:** Kullanıcıların PDF dosyalarını frontend arayüzünden seçip sunucuya yüklemesi gerekir. Bunu gerçekleştirmek için iki temel yöntem vardır: Birincisi, frontend'de doğrudan FastAPI'nin dosya yükleme endpoint'ine bir HTTP POST isteği göndermektir (örn. Axios veya fetch ile). Bu durumda FastAPI, `UploadFile` gibi bir mekanizma ile dosyayı alır ve işlemler. Bu yaklaşımda **CORS** yapılandırmasına dikkat edilmelidir; FastAPI ve frontend farklı origin'lerdeyse (alan adı/port), FastAPI uygulamasında CORS izinleri (örn. `FastAPI(cors_origin_allow_all=True)` veya belirli origin izni) tanımlanmalıdır. İkinci yöntem, Next.js gibi bir framework'ün **API route** özelliğinden yararlanmaktır. Next.js kendi backend route'larını tanımlamamıza izin verdiği için, bir `/api/upload` route'u yazıp, gelen dosyayı alıp FastAPI'ye sunucu-tarafından iletmek (proxy yapmak) mümkündür. Bu sayede tarayıcı sadece Next API ile konuşur ve CORS sorunu olmadan Next sunucusu arka planda FastAPI ile haberleşir. Ancak bu, ek bir adım olduğundan genelde doğrudan istemciden FastAPI'ye göndermek daha verimlidir. Dosya yükleme boyutları büyük olabilir; bu yüzden hem istemci hem sunucu tarafında **streaming upload** (parça parça yükleme) desteklenmelidir. Tarayıcı tarafında `<form>` veya AJAX ile çok parçalı (multipart) gönderim yapılırken, sunucu tarafında Starlette altında dosyalar otomatik olarak temp olarak stream edilerek yazılır – bu sayede büyük PDF'ler de bellek taşmadan alınır. Next.js cephesinde, 13 sürümüyle birlikte gelen App Router ve React Server Components yaklaşımı, dosya upload için spesifik bir iyileştirme getirmese de, API route yerine sunucu aksiyonu (Server Action) gibi yenilikler ileride kullanılabilir.

- **Veri İletimi ve SSR entegrasyonu:** Frontend ile backend'in etkileşiminin bir yolu, SSR aşamasında veri almaktır. Örneğin Next.js kullanıyorsak, bir sayfa için `getServerSideProps`

içinde FastAPI'den özet veya analiz sonuçları çekilebilir [62]. Bu durumda istek kullanıcının sayfa isteği sırasında yapılır, Node.js sunucu tarafında FastAPI'ye istek atar ve gelen JSON'u sayfa propslarına yerleştirir. Avantajı, sayfa HTML'i oluşurken verinin gömülü gelmesidir – kullanıcı tarayıcıda beklerken tek seferde sayfa ve sonuç yüklenir, SEO açısından da bu içerik indekslenebilir. Dezavantajı ise, FastAPI'deki işlemin süresine bağlı olarak sayfa yanıt süresinin uzamasıdır. Eğer PDF özetleme gibi işlemler birkaç saniye sürüyorsa, SSR bu süre boyunca bloklanacaktır. Bunu aşmak için, SSR yerine istemci tarafında (CSR) veriyi çekmek tercih edilebilir: Yani kullanıcı arayüzü yüklenir yüklenmez bir "Yükleniyor" göstergesi ile API çağrısı başlatılır, sonuç gelince gösterilir. Projemizde kullanıcı yüklediği PDF'in sonuçlarını anlık görmek isteyecektir; bu durumda SSR'den ziyade **kullanıcı etkilemine dayalı** API çağrıları mantıklı olacaktır. Örneğin kullanıcı "Analiz Et" butonuna bastığında bir POST isteğiyle FastAPI'ye PDF'in ID'sini veya içeriğini gönderir, yanıt alınınca frontende işler. Özetle, SSR tekniğini genellikle **sayfa ilk açılışında genel verileri göstermek** için kullanırız (ör. dashboard verilerini SSR ile çekmek SEO gerekmiyorsa bile hız için iyi olabilir), fakat kullanıcıya özel, anlık işlemler (dosya analizi gibi) için CSR API çağrıları daha esnek ve hızlı geri dönüşlü olacaktır. Next.js – FastAPI ikilisinde, getServerSideProps ile güvenli şekilde arka uçtan veri çekmek mümkün ve bu arka uç çağrıları sunucu tarafında yapıldığından tarayıcıya API anahtarları sızmaz, CORS sorunu yaşanmaz [62]. Bu da hassas veriler veya yetkilendirme gereken durumlar için bir avantaj.

- **Streaming Yanıtlar (Gerçek Zamanlı İletişim):** NLP işlemleri özellikle büyük metin özetleme, LLM tabanlı analiz gibi durumlarda zaman alabilir. Kullanıcı deneyimini iyileştirmek için yanıtın tamamı hazır olmasa bile parça parça iletmek isteyebiliriz. Bu amaçla **Server-Sent Events (SSE)** veya **WebSocket** kullanımı yaygındır. FastAPI, Starlette üzerine kurulu olduğu için SSE akışlarını veya WebSocket bağlantılarını destekler. Örneğin, özetleme yapan bir fonksiyon paragraf paragraf çıktısını üretiyorsa, FastAPI endpoint'ini `StreamingResponse` ile tanımlayıp her paragraf hazır olduğunda yield edebiliriz [63] [64]. Next.js tarafında SSE'yi yakalamak için tarayıcıda `EventSource` nesnesi kullanılabilir; bu sayede arka uçtan `text/event-stream` olarak akan veriler, frontend tarafında olaylar şeklinde yakalanıp bir sonuç alanında append edilebilir. Streaming yaparken dikkat edilmesi gereken nokta, arka uç ile ön uç arasındaki bağlantının kesilmemesidir – SSE HTTP üzerinden sürekli açık bir bağlantı kurar, FastAPI bu bağlantıyı tepkisel olarak açık tutar. Alternatif olarak **WebSocket** ile çift yönlü bir iletişim de kurulabilir; FastAPI `WebSocketRoute` desteğiyle bir websocket endpoint'i tanımlanabilir ve Next.js uygulaması içinde raw websocket (ya da kullanılabiliyorsa Next.js API route üzerinden proxy) ile bağlanabilir. WebSocket, SSE'ye göre çift yönlü iletişime imkan verse de, sadece sunucudan veri akıtmak için SSE daha hafif olabilir. Projemizde örneğin uzun bir PDF özetleme işlemi yapılırken, kullanıcının bekleme süresini daha iyi hale getirmek için "cümle cümle özet akışı" verilebilir. Bu durumda FastAPI şöyle çalışabilir: PDF'i parçalara ayırıp özetlerken her parçanın sonucunu SSE ile gönderir. Kullanıcı ara sonuçları görürken en sonunda tam sonuç oluşur. Bu teknik büyük dil modelleri ile çalışırken de kullanılır (OpenAI API'lerinin stream modunda yaptığı gibi). Kısacası, **gerçek zamanlı geri bildirim** gerektiğinde FastAPI + Next kombinasyonunda SSE en uygun yaklaşımlardandır ve literatürde de FastAPI üzerinde SSE ile LLM sonucu gönderme örnekleri mevcuttur [65] [66]. Next.js 13 itibariyle React 18'in streaming SSR özelliğini de kullanabiliyor; bu ise farklı bir seviyede – sunucunun ürettiği HTML parçalarını stream etmek. Eğer özet metni HTML olarak parça parça üretilebilirse, React bunu `suspsense` ile idare edip yanıtı stream edebilir. Ancak bu oldukça ileri bir senaryo; genelde daha basit olan SSE/WebSocket yolunu tercih etmek mantıklı.

- **Asenkron Görev İşleme ve Kuyruklar:** PDF yüklenmesi sonrası yapılan NLP analizleri (ör. dil kontrolü, özetleme) bazen birkaç saniyeden uzun sürebilir veya yoğun CPU kullanabilir. Bu durumda web sunucusunun cevabı beklerken bloke olmaması ve yüksek eşzamanlı istek geldiğinde birbirini etkilememesi için, **arka plan görev kuyrukları** kullanmak iyi bir çözümdür.

Python ekosisteminde Celery + RabbitMQ/Redis gibi araçlar bu amaçla yaygın kullanılır. Mimaride bu, FastAPI'nin bir isteği aldıktan sonra görevi bir kuyruk sistemine atıp anında 202 Accepted benzeri bir yanıt döndürmesi, daha sonra iş tamamlanınca sonucu bir yerde hazır etmesi şeklinde olabilir. Frontend tarafında ise kullanıcıya "işleniyor" durumu bildirildikten sonra sonuç hazır olduğunda bir bildirim (polling ile kontrol veya WebSocket ile push) yapılabilir. Orta ölçekli projelerde Celery kullanmak biraz komplekslik eklese de, ölçeklendirme gerektiğinde faydalıdır. Örneğin, birden fazla işçi süreç PDF'leri sırayla işler ve böylece FastAPI ana süreçleri hemen yanıt verip yeni istekler almaya devam eder. Next.js ile entegrasyonda, polling basit bir yaklaşımdır: Özetleme isteği yaptıktan sonra her 5 saniyede bir frontend sonuç endpoint'ini kontrol eder, hazırsa gösterir. Daha sofistike olarak SSE akışını direkt arka plan işçiden yapabiliriz veya WebSocket ile "tamamlandı" sinyali gönderebiliriz. **Streaming output** kullanıyorsak belki arka plan gerekmeden de kullanıcı beklerken ilerlemeyi görmüş olur. Celery gibi araçlar daha çok **yüksek hacimli paralel işler** için gereklidir. Eğer sistem aynı anda birçok PDF alacaksa ve her biri ağır hesaplamaysa, Celery + birkaç worker süreci ile iş kuyruğa alınır. FastAPI tarafında `BackgroundTasks` özelliği de küçük ölçek için basit bir alternatif olabilir (istek sonunda otomatik bir fonksiyonu arka planda çalıştırabiliyor). Ayrıca dosya işleme sırasında sunucunun yanıt süresi çok uzayacaksa, tarayıcının bekleme sınırlarına takılmamak için (ör. 30 saniye+) arka plan yaklaşımı kaçınılmaz olur. Kısacası, **asenkron görev dağıtımı** ihtiyacını belirlerken ortalama işlem süresi ve eş zamanlı kullanıcı sayısını göz önüne almalıyız. Yüksek throughput hedefleniyorsa, Next.js + FastAPI yapısında **Celery/RQ** gibi kuyruk sistemleri entegre edilmesi önerilir [67]. Düşük trafikli ve her isteğin birkaç saniye sürdüğü durumda ise, SSE ile akış sağlamak veya isteği açık tutmak (timeout'ları ayarlayarak) yeterli olabilir.

- **Güvenlik ve Diğer Entegrasyon Detayları:** Frontend-backend ayrık yapıda dikkat edilmesi gereken bir diğer nokta, **yetkilendirme ve kimlik doğrulama** bilgilerinin paylaşımıdır. Örneğin bir JWT kullanılıyorsa, Next.js istek yaparken token'ı ya API route üzerinden arka uca iletmeli ya da tarayıcıdan direkt gönderiyorsa CORS header'ları ile birlikte sağlamalıdır. Next.js bunu getServerSideProps içinde yaparsa, token sunucu tarafında güvenli kalır. Ayrıca upload edilen PDF'lerin saklanması (Disk'e mi yazılacak, S3 gibi bir buluta mı atılacak) konusu da entegrasyonun parçasıdır. Küçük ölçek için FastAPI gelen dosyayı işle ve sil şeklinde çalışabilir; daha büyük ölçek için dosya servislerini entegre etmek gerekebilir.

Özetle, Next.js gibi güçlü bir frontend ile FastAPI gibi esnek bir backend'in entegrasyonu, doğru tasarlandığında oldukça **verimli ve kullanıcı dostu** bir sistem sağlar. SSR ile hızlı ilk render, CSR ile interaktif işlemler, SSE/WebSocket ile gerçek zamanlı güncellemeler, arka plan görev kuyrukları ile ölçeklenebilirlik kombinasyonu, modern web uygulamalarının ihtiyaç duyduğu her şeyi kapsar. Nitekim gerçek dünya örneklerinde de Next.js + FastAPI ikilisinin sıkça kullanıldığı, **frontend Vercel'de, backend Docker ile sunularak** başarılı sonuçlar alındığı bilinmektedir [68] [51]. Bu yapıda bir ters proxy (NGINX veya Traefik) kullanarak aynı alan adı altında istekleri yönlendirmek de pratik bir çözümdür (örn. `myapp.com/api/*` isteklerini FastAPI'ye, kalanlarını Next uygulamasına iletmek). Doğru yapılandırıldığında kullanıcı, ön yüzde sorunsuz PDF yükleyip analiz sonuçlarını anlık alabilir; arka planda hangi teknolojilerin konuştuğunu bile anlamaz.

## NLP İşlemleri İçin En Verimli Modeller ve Araçlar (TR/EN Dil Desteğiyle)

Projede PDF içeriği üzerinde üç temel doğal dil işleme görevi planlanıyor: **yazım hatası tespiti**, **anlam/dilbilgisi hatası tespiti** ve **metin özetleme**. Bu görevler için mevcut araçları değerlendirirken, hem

Türkçe hem İngilizce dil desteği, çıktı kalitesi, çalıştırma verimliliği ve entegrasyon kolaylığı gibi kriterler göz önüne alınmalıdır.

- **Yazım Hatası Tespiti (Spell Checking):** En temel gereksinim, metindeki yanlış yazılmış kelimeleri saptamaktır. İngilizce için birçok iyi çözüm vardır; Türkçe için de özelleşmiş araçlar bulunur. Öne çıkan iki yaklaşım: **kural tabanlı sözlük + dilbilgisi araçları** ve **yapay zeka tabanlı modeller**. Kural tabanlı tarafta en popüler araç **LanguageTool**'dur. LanguageTool, 20'den fazla dili destekleyen açık kaynak bir dil denetleme aracıdır [69]. İçinde hem sözlük tabanlı yazım denetimi yapar hem de tanımlı dilbilgisi kurallarına göre hataları yakalar. İngilizce için LanguageTool'da 6000'den fazla kural bulunur ve oldukça başarılı sonuç verir [70]. Ancak **Türkçe desteği** LanguageTool tarafında sınırlıdır – projeye Türkçe kurallar eklenmesi topluluk desteğine bağlı kalmıştır ve şu anki durumda güncel Türkçe kuralları eksiktir [71]. LanguageTool geliştiricileri, Türkçe desteğinin eski sürümlerde kısıtlı kaldığını ve katkı gelirse genişletilebileceğini belirtmişlerdir [71]. Dolayısıyla Türkçe için LanguageTool kullanılırsa, sadece temel yazım denetimi (belki hunspell sözlük ile) yapabilecektir, ileri gramer hatalarını yakalaması beklenmez. Türkçe özelinde açık kaynak en güçlü araç **Zemberek** kütüphanesidir. Zemberek, Java dilinde geliştirilmiş kapsamlı bir Türkçe doğal dil işleme kütüphanesidir ve içinde yazım denetimi, türetme, heceleme gibi modüller barındırır [72]. Zemberek'in yazım denetimi performansı oldukça yüksektir: Yapılan bir testte 71 bin kelimelik bir Türkçe roman üzerinde saniyede **75.000 kelime**yi denetleyebildiği rapor edilmiştir [73]. Bu rakam, aracın verimliliğinin altını çizmektedir (karşılaştırma için, sıradan bir yazım denetleyici bu hızlara ulaşmakta zorlanır). Zemberek ayrıca yanlış birleşik yazımlar, klavyede Türkçe karakter kullanılmaması durumları gibi Türkçe'ye özgü hataları da tespit edebilir ve düzeltme önerileri sunar [74] [75]. Projemizde Zemberek'i kullanmak, Java tabanlı olduğu için biraz uğraş gerektirebilir (Python'dan JNI veya bir REST servis aracılığıyla kullanmak gibi). Alternatif olarak, **PySpellChecker** gibi Python kütüphaneleri de basit yazım kontrolü yapabilir ama bunlar dilin kurallarını bilmez, sadece sözlük karşılaştırması yapar. İngilizce için endüstri standardı araç **Hunspell** (OpenOffice/LibreOffice'te kullanılan) ve **LanguageTool** kombinasyonudur. LanguageTool, İngilizce'de sadece yazım değil yüzlerce dilbilgisi kuralını da yakalar (ör. article hataları, yanlış fiil çekimi vb). Ayrıca LanguageTool'un bir **Python API**'si (language_tool_python) bulunmaktadır, kendi sunucusunu başlatıp metinleri kontrol edebilir [76]. Özetle: *İngilizce yazım/dilbilgisi kontrolü* için **LanguageTool** güçlü bir ücretsiz seçenektir (Grammarly gibi ücretli servislerin açık kaynak karşılığı gibidir). *Türkçe yazım denetimi* için **Zemberek** birinci tercih olmalıdır; dakikada milyonlarca kelimeye kadar ölçeklenebilir hızıyla verimli çalışır [73]. Eğer daha gelişmiş bir yol istenirse, **yapay zeka tabanlı** (n-gram veya dil modeli temelli) düzelticilere bakılabilir. Örneğin **Google'in Seq2Seq düzeltme modelleri** veya **T5 tabanlı gramer düzeltme** modelleri mevcuttur. Hugging Face üzerinde "yeniguno/mbart50-turkish-grammar-corrector" gibi bir model, Türkçe cümle düzeltme amacıyla mBART50 modelini ince ayar yapmıştır [77]. ~600 milyon parametreli bu model, eğitimli olduğu hataları (özellikle eğitim verisindeki yapay hataları) düzeltebilmektedir. Ancak böyle bir modeli çalıştırmak oldukça kaynak ister (tercihen GPU). İngilizce için de HuggingFace'de **T5** veya **GPT-2** tabanlı Gramformer gibi modeller bulunur. Bu modellerin çıktıları bazen kural tabanlı araçlardan daha akıllı olsa da, yanılgıları olabilir ve çalıştırma maliyetleri yüksektir. Projemizde çevrimdışı/yerel bir çözüm istiyorsak ve kalitede en üst seviyeyi hedefliyorsak, **OpenAI'nın GPT-4 API**'ını da bir seçenek olarak düşünebiliriz – hem Türkçe hem İngilizce metinlerde mükemmele yakın düzeltme ve özet yapma kapasitesine sahiptir. Fakat bu bir bulut servisi olduğu için veriyi üçüncü partiye göndermek ve kullanım başına ücretlendirme gibi konular var, ayrıca hız olarak küçük modellerden yavaş olabilir.

- **Dilbilgisi ve Anlam Hataları (Grammar & Semantic Errors):** Yazım hatasından daha zor olan kısım, doğru yazılmış ama anlamca yanlış veya dilbilgisi bozuk cümleleri tespit etmektir. İngilizce için bu genelde "grammar check" olarak bilinir. Türkçe'de "anlam hatası" muhtemelen yanlış ek

kullanımı, anlamsız cümle veya bağlam hatalarını ifade ediyor. **LanguageTool**, İngilizce'de dilbilgisi hatalarını yüzlerce kural ile yakalar (örn. "a/an" kullanım hataları, tekil-çoğul uyumsuzluğu, yanlış kelime seçimi gibi). Türkçe'de ise kural yazımı az olduğundan, örneğin devrik cümle veya ek uyumsuzluğu gibi hataları tespit etmekte yetersiz kalabilir. Zemberek kütüphanesi, Türkçe'de **morfolojik analiz** yaparak bir cümlenin yapısını çözebilir ama dilbilgisel olarak doğru olup olmadığına dair sınırlı çıkarım yapar. Örneğin "geliyorum" vs "geliyourm" yazım hatasını Zemberek bulur, ancak "geliyor mu" ayrı yazılmalıydı gibi bir hatayı kural tanımlı değilse bulamayabilir. Bu noktada iki yaklaşım öne çıkıyor: (1) **Kendi kural setimizi geliştirmek:** Örneğin Türkçe'de sık yapılan belirli hatalar için (de/da'nin ayrı yazılması, ki bağlacının kullanımı, mi soru ekinin ayrılığı, ek-fiillerin (yapıyor musun vs yapıyormusun) bitişik yazım hatası gibi) küçük bir kural motoru yazılabilir veya LanguageTool'a kural eklenebilir. (2) **Yapay zeka ile hata tespiti:** Bu, bir cümleyi bir dil modeliyle değerlendirip "doğru mu yazılmış?" sorusunu sormak demektir. Mesela GPT-3.5'e "Bu cümlede dilbilgisi hatası var mı?" diye sorarsanız, yüksek doğrulukla tespit edecektir. Ancak yerel çalıştırılacak bir modelle bunu yapmak zor. HuggingFace'de **Türkçe Gramer Hata Düzeltme (GEC)** için akademik çalışmalar mevcut [78]. Örneğin bir çalışma, Türkçe sentetik hatalarla zenginleştirilmiş bir GEC veri kümesi oluşturup bir T5 modelini eğitmiş [79]. Bu tür modeller, cümleyi girdip düzeltilmiş halini çıkarıyor. Hata tespiti için ise orijinal ve düzeltilmiş cümleyi kıyaslamak gerekiyor. Projemizde gerçek zamanlılık kritik değilse, belki cümle cümle GPT API'ye göndermek bile düşünülebilir – ama bu mahremiyet ve maliyet sorunları doğurabilir. Özetle, **İngilizce** metinler için *dilbilgisi kontrolü*ne en pratik çözüm LanguageTool ya da Grammarly API (ücretli) gibi servislere başvurmaktır. **Türkçe** için ise *dilbilgisi/anlam hatası* yakalamada en iyi yaklaşım şimdilik **yapay zeka destekli** olandır; zira kural bazlı sistemimiz zayıf. Örneğin, bir cümleyi Zemberek morfolojik çözücü ile inceleyip imla hatası yoksa, bir de OpenAI'nın ücretsiz bir modeline (örn. davinci) sorarak hata buldurtulabilir. Ya da "yeniguno/mbart50-turkish-grammar-corrector" modelini indirip çalıştırarak cümlenin önerilen doğrusunu alıp orijinalle kıyaslayabiliriz. Bu model, Türkçe eğitildiği için *bazı* dilbilgisi hatalarını yakalar ama hepsini değil. Kalite açısından GPT-4 bu konuda çıtayı belirlemiş durumda, ancak yerelde çalışmaz. Belki ileride daha küçük açık modeller (LLaMA 2 13B gibi) Türkçe konusunda iyi hale gelir ve onları entegre edebiliriz.

- **Metin Özetleme:** Projenin en dikkat çekici özelliği PDF özetleme olduğuna göre, burada doğru teknoloji seçimi kritik. İki dilde de iyi özet çıkarabilmek gerekir. Seçenekler: **Açık kaynak özetleme modelleri** veya **API tabanlı güçlü modeller**. Açık kaynak tarafında, *İngilizce* için Google'ın **PEGASUS** modeli, Facebook'un **BART** veya **T5** tabanlı modelleri popülerdir. Bu modeller haber makaleleri gibi veri setlerinde eğitilmiş ve oldukça iyi özetler üretebilir. Türkçe için yakın zamanda hazırlanan veri kümeleri ve modeller var: Örneğin **MLSUM (Multi-Lingual Summarization)** veri kümesinin Türkçe bölümü kullanılarak ince ayar yapılmış **mBART50** ve **mT5** modelleri mevcut [80] [81]. Ali Safaya ve arkadaşlarının "Mukayese: Turkish NLP Strikes Back" çalışmasında, mBART-large-50 modeli Türkçe haber özetlemede ROUGE-2 skoru ~%34'e ulaşmış, bu oldukça iyi bir değer [81]. Bu modeli HuggingFace'den indirip kullanmak mümkün (611 milyon parametre, ~2.4GB boyutunda). Eğer sunucu tarafında GPU yoksa bu boyuttaki bir model CPU'da özetleme yaparken yavaş olacaktır (birkaç yüz kelimelik paragrafı özetlemek belki 8-10 saniye sürebilir). Daha küçük modeller de var: Örneğin "ozcangundes/mt5-small-turkish-summarization" (300MB civarı, daha düşük kalite ama hızlı) [82]. *İngilizce* için ise **t5-base** veya **distilbart-cnn-12-6** gibi küçültülmüş özetleme modelleri hızlı sonuç verebilir. Bu modelleri entegre etmek için HuggingFace **Transformers** kütüphanesini kullanmak yeterli; pipeline arayüzü ile birkaç satırda özet alınabilir. Ancak boyut ve bellek kullanımına dikkat etmek lazım – mBART-large gibi model bellekte ~2GB yer tutar, CPU'da dakikada birkaç özet anca yapabilir. Eğer kullanıcılar çok sayıda büyük PDF yükleyecekse, **daha güçlü bir yaklaşıma** ihtiyaç duyabiliriz. Bu noktada **bulut tabanlı API'ler** devreye giriyor. Örneğin, OpenAI'nın GPT-3.5 ve GPT-4 modelleri hem Türkçe hem İngilizce metinleri başarılı şekilde özetleyebiliyor (hatta paragraf başlıkları

çıkarma, maddeleme gibi kontroller de sağlanabilir). OpenAI API'ye PDF içeriğini gönderip "bana X dilinde 3 paragraflık özet ver" gibi bir istemle mükemmele yakın bir özet almak mümkün. Kalite açısından GPT-4, bugünün açık modellerinden ileridedir. Ancak dezavantajları: **Maliyet** (her token başına ödeme), **hız** (yoğun saatlerde gecikme olabilir) ve **veri gizliliği** (içerik üçüncü taraf sunuculara gidiyor). Eğer özetlenecek PDF'ler hassas kurumsal belgeler ise, API kullanımı sakıncalı olabilir. Bu durumda veri merkezine kapanmış bir model daha iyi. Orta yol olarak **HuggingFace Inference API** veya **Cohere, AI21** gibi alternatif API'ler de düşünülebilir ancak benzer endişeler onlar için de geçerli. Teknik olarak bir diğer yaklaşım da, PDF'den *önemli cümleleri seçen* klasik özetleme yöntemleri (textrank, vs.) kullanmaktır – bunlar hızlıdır fakat girdi çıkarmalı (abstraktif) özet kadar akıcı özet vermezler. Textrank gibi algoritmalar cümlenin skoruna göre en önemli 2-3 cümleyi seçer; bu bazen iş görebilir ama istenen özet niteliğini sağlamayabilir.

**TR/EN dil desteği ve kalite değerlendirmesi:** Türkçe dil işleme araçları İngilizce'ye kıyasla daha sınırlı olsa da son dönemde önemli gelişmeler oldu. Yazım ve dilbilgisi için kural tabanlı araçlarda İngilizce üstün (LanguageTool İngilizce'de çok kapsamlı), Türkçe'de ise Zemberek gibi bir uzmanlık kütüphanesi mevcut ancak dilbilgisi kuralları için açık kaynak projeler henüz yeterince değil. Kalite açısından, Türkçe metinlerde hataları bulmak İngilizce'den zor çünkü Türkçe aglütine yapısı hataların çeşitliliğini artırıyor. Örneğin İngilizce'de tekil-çoğul uyumsuzluğu bariz bir hatayken, Türkçe'de eklerin yanlış kullanımı incelikli olabiliyor. Bu nedenle, Türkçe **anlam hatalarını tespit** görevinde GPT-4 gibi büyük modellerin bariz üstünlüğü var – hem dilin nüanslarını anlıyor hem de öneri verebiliyor. Açık kaynak cephesinde 2023 itibariyle 7-13 milyar parametreli açık modellerin (LLaMA türevleri vs) Türkçe anlama yetenekleri fena değil ancak hala belirli hataları kaçırabilirler [83] [84]. Özetle, *en yüksek kaliteye* odaklanırsak, şöyle bir yaklaşım makul: **İngilizce** için LanguageTool + özetlemede PEGASUS (veya GPT-3.5); **Türkçe** için Zemberek + özetlemede mBART-large veya GPT-3.5, ve dilbilgisi için belki ufak bir GPT-3.5 isteği. Fakat proje tamamen yerel çalışacaksa GPT kullanamayız. Bu durumda bahsettiğimiz açık modelleri entegre edeceğiz. Bunların verimliliğini sağlamak için de **model boyutu ve donanım uygunluğu** göz önünde tutulmalı. Örneğin 600M parametrelik bir özetleme modeli CPU'da yavaş kalırsa, belki özetlemeyi kısmen kullanıcı inisiyatifine bırakabiliriz (PDF çok uzunsa önce metni bölüp özetlemek gibi).

Ek bir not: PDF yükleme ve işleme sürecinde **OCR ihtiyacı** olabilir. Eğer PDF görüntü tabanlıysa, içinde metin yoksa önce OCR ile metne çevrilmeli (Tesseract gibi). Bu da bir miktar zaman ve entegrasyon demek. OCR sonrası çıkan metinlerde yazım hatası daha fazla olabilecektir; dolayısıyla yazım denetimi modülü OCR hatalarını da kısmen temizlemeye yardımcı olabilir.

Sonuç olarak, NLP görevlerinde **en verimli araç kombinasyonu** şöyle özetlenebilir:

- *Yazım ve temel dilbilgisi kontrolü:* **LanguageTool** (İngilizce için kapsamlı, Türkçe için sınırlı kurallarla) + **Zemberek** (Türkçe imla için hızlı ve kapsamlı) birlikte kullanımı. İngilizce metinleri doğrudan LanguageTool'a verebiliriz, Türkçe metinlerde önce Zemberek ile bariz hataları bulup düzeltme önerileri alınabilir, ardından geri kalanını (örn. noktalama, ekstra boşluk vs) LanguageTool halleder. Bu ikisi açık kaynak ve offline çalışabilir. Zemberek'in rapor edilen hızı saniyede 75 bin kelime olduğundan büyük belgelerde bile anlık sonuç üretebilir [73]. LanguageTool da Java tabanlı olduğundan benzer şekilde optimize çalışır (kendi dilinde). Performans testlerinde, Zemberek'in yazım denetleme modülünün pratikte işlem süresi PDF işleme hattında ihmal edilebilir düzeyde olacaktır.

- *Gelişmiş dilbilgisi/anlam denetimi:* **Açık kaynak model** olarak mümkünse bir Türkçe Grammatical Error Correction (GEC) modeli entegre etmek verimli olur. Örneğin "nezahatkorkmaz/grammar-correction-t5-small" (Türkçe için küçük bir T5 modeli) veya yeniguno'nun mBART50 modeli gibi. Bu model cümleleri düzeltilmiş haliyle vereceği için, bir cümle orijinali ile model çıktısı aynıysa

hata yok, farklıysa modele göre hata var demektir. Bu şekilde kullanıcıya “Şu cümle daha doğru olabilir: …” gibi öneriler sunulabilir. Yine İngilizce için de benzer bir Gramformer model çalıştırılabilir. Bu modellerin hızını artırmak için **8-bit quantization** yapılabilir (HuggingFace int8). Eğer bu karmaşa istenmezse, belki bu özelliği MVP’de basit tutup sadece kural tabanlı yakalamalar yapmak, ileride model entegre etmek de düşünülebilir. Zira zaman kısıtında hepsini birden en iyi şekilde yapmak zor olabilir.

- *Metin özetleme:* **mBART-large Türkçe özetleme modeli** (611M parametre) makul bir seçim, zira haberlere dayalı eğitimi genel dilde de işe yarayacaktır. Bu modelin kalite metrikleri yüksek (ROUGE-L ~41) [81] . İngilizce için de **PEGASUS (Google)** veya **BART CNN modeli** kullanılabilir. Aslında mBART-50 modeli çoklu dilli bir model olduğundan İngilizce’yi de içerir. Fakat en iyisi, **iki dil için ayrı modeller** kullanmak olabilir, çünkü her dil için özelleşmiş modeller genelde daha iyi sonuç verir. Örneğin İngilizce için Facebook BART-large-CNN özetleme modeli sektörde kendini kanıtlamıştır. Bu model de ~400M parametre civarı. Özetleme modellerini çalıştırırken, özet uzunluğunu kontrollü tutmak (max_length parametresi ile) ve gerekirse bölerek özetlemek gerekebilir. Bir 50 sayfalık PDF’i tek seferde özetlemeye çalışmak herhangi bir model için zordur; bunun yerine her bir bölümü ayrı özetleyip sonra birleşik bir özet yapmak gibi stratejiler uygulanabilir.

- *Kalite & Geri Bildirim:* Elde edilen özet ve düzeltme sonuçlarının kullanıcıya sunumu da önemli. Örneğin dilbilgisi hataları bulunduğunda, bunların altını çizip üzerine gelince öneri göstermek (Office Word tarzı) iyi bir kullanıcı deneyimidir. Bunu yapmak için frontend’de ilgili cümleleri işaretlemek gerekir. Model bazlı yaklaşımlarda hangi kelimenin hatalı olduğunu bulmak zor olabilir (çünkü model sadece düzeltir, neyi düzelttiğini açıklamaz). Kural tabanlı LanguageTool ise tam olarak hangi karakter aralığında hata olduğunu verir, bu kullanılabilir.

Sonuç olarak, projemiz için **en verimli NLP çözümü** kağıt üzerinde şöyle görünmektedir: **Türkçe** için Zemberek + (imkan dahilinde) Türkçe GEC modeli + mBART özetleme; **İngilizce** için LanguageTool + belki GingerIt (basit bir grammar corrector) + PEGASUS özetleme. Bu kombinasyon açık kaynak araçlarla hedeflenen işlevleri yerine getirir. Hepsini Python’da entegre edebiliriz: Zemberek için bir Python wrapper veya subprocess çözümü, LanguageTool için `language_tool_python`, HF modelleri için `transformers` pipeline. Bu sayede FastAPI backend’i tüm bu araçları kullanarak gelen PDF’in metnini işler, hataları ve özetleri çıkarır, sonuçları frontend’e JSON olarak iletir. Performans olarak, bu işlem birden fazla alt adımı içerdiği için optimize edilmesi gereken yerler olabilir (örneğin model yüklemeleri baştan yapılmalı, her istek için yeniden yüklenmemeli). Tek bir özetleme ~5-10 saniye sürebilir model boyutuna göre, bunu kabul edilebilir kılmak adına kullanıcıya streaming ile cümle cümle göstermek veya en azından bir ilerleme çubuğu vermek önemlidir.

## En Verimli Tam Yığın Yapı Önerisi (Performans, Esneklik ve Geliştirici Deneyimi Açısından)

Yukarıdaki analizler ışığında, proje için **en uygun tam yığın teknoloji yığını** ve mimarisi şu şekilde önerilebilir:

- **Frontend:** Next.js (React) tercih edilmesi, mevcut kullanım ve ekibin muhtemel React bilgisi göz önüne alındığında mantıklı görünüyor. Next.js, SSR desteği sayesinde projenin PDF analiz sonuçlarını SEO’ya uygun şekilde sunma opsiyonunu açık tutar (her ne kadar kullanıcıya özel içerik olsa da, belki genel tanıtım sayfaları SSR ile optimize edilebilir). Ayrıca Next.js 13 ile gelen React 18 özellikleri (Streaming, Suspense) ileride kullanılabilir bir avantaj sağlayabilir. En önemlisi, Next.js’in geniş eklenti ekosistemi (UI kütüphaneleri, hazır komponentler, vb.) ile geliştirme süresi

kısalacaktır. Geliştirici deneyimi açısından Next.js olgun bir tercih olup, ekipte yeni biri dahil olsa bile öğrenme materyali boldur. Performans olarak, SvelteKit gibi rakipler daha hızlı ilk yükleme sunsa da Next.js + React kombinasyonu iyi optimize edildiğinde kullanıcı açısından yeterince hızlı olacaktır. Özellikle PDF yüklendikten sonra sonuç sayfasında çok ağır bir arayüz olmayacağı için React'in getirdiği ekstra yük sorun olmayacaktır. Ayrıca Next.js ile siteyi Vercel gibi bir platforma kolayca dağıtmak mümkündür, bu da CI/CD ve ölçeklendirme için kolaylık sağlar. **Alternatif:** Eğer ekipte Svelte konusunda istek ve uzmanlık varsa, SvelteKit de frontend için düşünülebilir; özellikle uygulamanın sadece birkaç sayfalık bir arayüzü varsa SvelteKit ile daha az JS göndererek daha hızlı bir etkileşim sağlayabilirsiniz [8]. Fakat Next.js'in halihazırda projede kullanılıyor oluşu ve ekosistem avantajları düşünüldüğünde, *en risksiz ve dengeli tercih Next.js olmaya devam ediyor*. Nuxt.js ise Vue ekosistemini sevenler için benzer bir çözüm olurdu; ancak proje başlangıçta Next.js ile yazıldığına göre, Nuxt'a geçiş ek bir öğrenme eğrisi getirir, bu da gerekmez.

- **Backend:** Python FastAPI backend'i, proje gereksinimleri için **en makul seçenek** olarak öne çıkıyor. En büyük gerekçe, NLP işlemlerinin Python ekosisteminde çok daha rahat yapılabilmesi. Yukarıda belirttiğimiz gibi dil modelleri, Zemberek entegrasyonu, vs. Python'da ya doğrudan mevcut ya da Python'dan çağrılabilir durumda. Node.js ile de bazı AI işlemleri mümkün ama genelde Python kadar çeşitlilik ve olgunluk yok. Go ile ise neredeyse tüm bu işleri sıfırdan bağlamak gerekir. FastAPI performans konusunda da Python dünyasında lider ve asenkron olduğu için I/O beklemelerinde (dosya okuma, HTTP çağrıları vs) minimum gecikme sağlıyor. Bu projenin kullanıcı sayısı muhtemelen sınırlı (bir SaaS değil de bir araç gibi görünüyor), dolayısıyla FastAPI'nin sağlayacağı birkaç bin istek/s kapasite pekâlâ yeterli olacaktır. Yine de, ileride yüksek trafik olması durumunda dahi FastAPI yatay olarak ölçeklendirilebilir (örn. Kubernetes'de 5 replika FastAPI pod'u vs). Node.js/NestJS backend kullanımı, tek dilde tutarlılık getirirdi ancak model entegrasyonu konusunda ya Python servislerine mecbur kalınırdı ya da OpenAI API gibi harici servislere bağımlı olunurdu. Bu da ya geliştirme karmaşıklığını ya da maliyeti artırırdı. **Verimlilik açısından** da, Python kodunun içinde asıl yükü taşıyan kütüphaneler genelde C/C++ ile optimize edilmiş durumdadır (ör. PyTorch tensör işlemleri, HuggingFace'in BLAS çağrıları, Zemberek Java kodu vs). Yani Python burada sadece yapıştırıcı dil olacak; ağır işlemler ya alt düzeyde verimli biçimde çalışacak ya da GPU kullanacak. Dolayısıyla Python'un kendisinin yavaşlığı çok belirleyici olmayacaktır. FastAPI'nin artı puanlarından biri de geliştiricinin hızlı prototip çıkabilmesi – otomatik dokümantasyonla API'yi test etmek, Pydantic ile veri tiplerini tanımlamak ve doğrulamak, dependency injection sistemi ile modüler yapmak gibi özellikler, bu projede güvenli ve hatasız geliştirmeyi kolaylaştırır. Unutulmamalı ki, projenin odak noktası NLP işlemlerinin başarısı; Python bunu sağlamada en olgun platformdur. Nitekim bir kaynakta da belirtildiği gibi, **AI/ML tabanlı servisler geliştirirken FastAPI tercih etmek, Node.js'e kıyasla daha uygun olabilir** [85] [86]. Node.js ise gerçek zamanlı, hafif işlemlerde önde gider. Bizim senaryomuz bir AI servisiyse, FastAPI'nin seçilmiş olması doğru yöndedir.

- **Tam Yığın Entegrasyon ve Mimari:** Next.js + FastAPI kombinasyonunu nasıl konumlandıracağımıza gelirsek, önerilen mimari şu şekildedir: Frontend Next uygulaması, örneğin Vercel ya da Netlify gibi bir ortama veya kendi Nginx sunucusunda static build olarak kurulabilir. FastAPI backend ise bir Docker konteyner içinde Uvicorn/Gunicorn ile (ör. 4 worker) çalıştırılabilir. Araya bir ters proxy koymak (Nginx) hem statik dosyaları servis etmek hem de API isteklerini FastAPI'ye yönlendirmek için faydalı olacaktır. Kullanıcı tarayıcısından gelen istekler şöyle akar: `GET /` (anasayfa) -> Next.js (SSR veya statik), `POST /api/analyze` (PDF yükle & analiz isteği) -> FastAPI (JSON sonuç döner), `GET /api/stream` (SSE endpoint diyelim) -> FastAPI (akış yanıtı döner). Bu dağıtık yapıyı alan adı bazında birleştirmek CORS sorununu kökten çözer (aynı domain altında proxylandikleri için). Örneğin, **NGINX config** şu şekilde olabilir: `location / -> proxy_pass http://nextjs;` ve `location /api/ -> proxy_pass http://fastapi;`. Bu durumda Next uygulaması sadece ön yüz sayfalarını sunar, tüm `/api`

yoluna gelenleri FastAPI'ye iletir. Bu tür bir mimari, birçok kurumsal projede kullanılır ve oldukça esnektir. Ayrıca Docker kullanıldığı için FastAPI konteynerine gerekli model dosyaları, Zemberek vs. eklenip tek bir imaj haline getirilebilir. Bu imaj kubernetes üzerinde de çalıştırılabilir ileride. Next.js tarafı statik build alınıp bir CDN'e konacaksa (Vercel gibi), o zaman FastAPI farklı bir subdomain'de çalışabilir (`api.proje.com` gibi), o durumda da CORS'u açmak gerekir. Bu mimaride ölçeklendirme de basittir: Trafik artarsa FastAPI konteyner sayısı arttırılır veya GPU ihtiyacı olursa GPU'lu bir sunucuda çalıştırılır. Next.js tarafı zaten statik servis edilebildiği için ölçeklemesi kolaydır. Geliştirici deneyimi açısından da bu ayrım güzeldir: Frontend geliştiricisi rahatça Next.js ile çalışır, tasarıma odaklanır; Backend geliştiricisi FastAPI ile API ve AI mantığına odaklanır. Arada REST/JSON sınırı olduğundan sorumluluklar net ayrılır.

Performans odağında da bu kombinasyon oldukça dengelidir. Next.js sunucu tarafı Node.js üzerinde çalışsa da büyük kısmı I/O-bound işler yapacak (HTML render etmek vs). FastAPI ise CPU-bound AI işler yapacak. Bu iki yükü farklı makinelerde tutmak, birinin diğerini yavaşlatmasını önler. Örneğin aynı Node.js prosesi içinde NLP modelini çalıştırmaya kalksaydık (theoretically via TensorFlow.js), Node event loop'u bloklanabilirdi. Şimdi ise Node ve Python ayrı olduğundan paralel çalışabilirler. Bir kullanıcı PDF yüklerken Node tarafı beklemede kalmaz, Python arka planda işleyip bitince Node'a döner. Böyle asenkron bir mimari, **yük altında tepki verebilirlik (responsiveness)** açısından olumludur.

- **Ek Bileşenler:** Asenkron iş kuyruğu ihtiyacı duyulursa (çok kullanıcılı senaryoda) araya bir **Redis + RQ veya RabbitMQ + Celery** ekleyebiliriz. Bu durumda FastAPI istek alır almaz işi kuyruğa atar ve bir job id döner. Kullanıcı arayüzü bu id ile sonucu bekler. Worker (celery) bitirince sonuç Redis'te bir yere yazar, kullanıcıya SSE ile iletilir veya kullanıcı manuel sorgular. Bu mimari, sistem büyürse devreye alınacak bir opsiyon olarak kenarda tutulabilir. Yine bir **yük dengeleyici (load balancer)**, FastAPI örnekleri arasına konabilir.

- **Geliştirici Deneyimi Açısından:** Bu yığın hem popüler teknolojiler içerdiği için dokümantasyon, örnek proje, topluluk desteği boldur, hem de birbirinden bağımsız geliştirilebilir. Örneğin frontend tarafı dummy API cevaplarıyla geliştirme yapabilirken, backend tarafı Postman ile test edilebilir. TypeScript ve Pydantic şema tanımları sayesinde uçtan uca tür güvenliği de sağlanabilir (örn. istek/cevap modelleri tanımlanıp dökümantasyondan takip edilebilir). Bir başka artı, iki dil kullanılıyor olsa da ekip içinde yetkinlik dağılımına göre işler paylaşılabilir; ya da "full-stack" biri her iki tarafta da çalışabilir. Many-to-many ilişki yok, arayüz REST ile net ayrılmış.

- **Alternatif Tam Yığınlar Değerlendirmesi:** Tek dil kullanalım diye Node.js + (Next veya SvelteKit) + belki TensorFlow.js demek, kâğıt üstünde çekici ama pratikte NLP için sınırlayıcı. Zira Node'da Python kadar güçlü açık kaynak NLP aracı yok. Bir ihtimal, tüm işi OpenAI API'ye yıkıp Node.js arka uç kullanımı olabilirdi – yani Node backend sadece bir proxy olur, tüm dil işlemini OpenAI yapar. Bu durumda performans ve kalite yüksek olabilirdi (GPT-4 sayesinde), ancak sürekli bir API maliyeti ve bağımlılığı oluşurdu. Ayrıca hassas veriler OpenAI sunucularına gitmiş olurdu. Bu nedenle, daha **bağımsız ve maliyet etkin** bir çözüm olarak Python backend mantıklı. Go backend ise performans şahane olsa bile geliştirme hızı ve kütüphane eksikliği açısından çok efor gerektirirdi; projenin zaman ve kaynak kısıtları varsa uygun değil. Örneğin PDF parsing için Python'da PyMuPDF kullanmak varken, Go'da CGO ile Poppler sarmalamakla uğraşmak gerekebilirdi. Bu tip detaylar düşünüldüğünde, projenin gerektirdiği esneklik Python'da mevcut. Geliştirici deneyimine vurgu yaparsak, Python'daki **basitlik ve hızlı prototipleme** gücü göz ardı edilmemeli; NestJS'in enterprise yapısı belki bu boyuttaki bir proje için ağır bile kaçabilir (daha çok çok sayıda entity'li büyük sistemlerde parlıyor). Nitekim bir benchmark yazısında da "ekibiniz Python biliyorsa FastAPI'yi, JS biliyorsa NestJS'i seçin, gümüş kurşun yok" deniyor [85] [86] – bizim durumumuzda Python'a ihtiyaç bariz olduğundan FastAPI yolu isabetli.

**Sonuç:** En verimli tam yığın yapı, **Next.js + FastAPI** bileşimini, aralarında JSON/SSE iletişimi ile kurgulamak olacaktır. Bu kombinasyon, **performans** açısından yeterli hızı sağlar (ağır işlemler verimli kütüphanelerle yapılıyor; web tarafı SSR ile optimize), **esneklik** açısından her modülde en uygun aracı kullanma imkanı verir (frontend'de zengin React ekosistemi, backend'de güçlü Python NLP ekosistemi), **geliştirici deneyimi** açısından da üretkenliği yüksek tutar (tip güvenliği, otomatik dokümantasyon, tek komutla deploy gibi kolaylıklar). Ayrıca bu yığın ölçeklenmek istendiğinde yatay büyümeye hazırdır ve bulut ortamlarında yaygın olduğu için bakım maliyeti düşüktür. SvelteKit + FastAPI de benzer bir profil çizecektir ama Next.js'in yaygınlığı bir avantajdır. Node/NestJS + Python mikroservis şeklinde bir ayrım yapmak yerine, doğrudan FastAPI'yi ana backend yapmak mimariyi sadeleştirir.

Sonuç olarak, projenizin gerektirdiği PDF yükleme, Türkçe-İngilizce dil işlemleri, gerçek zamanlı çıktı gibi ihtiyaçlara en iyi cevap veren tam yığın çözüm **"Next.js (React) + FastAPI (Python)"** olarak değerlendirilebilir. Bu yığının üzerine, *LanguageTool* + *Zemberek* + *HuggingFace modelleri* ile NLP yetenekleri kazandırıldığında, sisteminiz hem teknik gerekçelerle temeli sağlam olacak hem de kullanıcılar için hızlı ve tutarlı bir deneyim sunacaktır. Yeni teknoloji seçenekleri cezbedici gelse de (örn. Go backend veya Svelte frontend), mevcut koşullarda **en dengeli tercih bu kombinasyondur** diyebiliriz. Zira hem geliştirme hızı, hem topluluk desteği, hem de yeterli performans bu seçenekle elde ediliyor. Gerçek dünyada da Hulu, Netflix gibi şirketlerin **Python tabanlı AI servislerini Node tabanlı frontendlere entegre ettiği** örnekler mevcuttur – benzer bir tasarım sizin projenizde de başarıyla uygulanabilir [87] [88] .

Bir cümleyle özetlemek gerekirse: *React/Next.js frontend + Python/FastAPI backend, üzerine açık kaynak NLP araçları kullanımı, bu proje için performans, esneklik ve geliştirici deneyimi boyutlarında en avantajlı tam yığın mimari olacaktır.* [85] [56]

---


[1] [3] [4] [5] [6] [7] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [41] [42] [43] [44] [45] [47] SvelteKit vs. Next.js: Which Should You Choose in 2025?
https://prismic.io/blog/sveltekit-vs-nextjs

[2] [32] [33] [34] [35] [36] Astro vs. Next.js: Features, performance, and use cases compared | Contentful
https://www.contentful.com/blog/astro-next-js-compared/

[8] Next.js vs SvelteKit vs Qwik: Best Framework in 2025
https://tymonglobal.com/blogs/next-js-vs-sveltekit-vs-qwik-best-framework-in-2025/

[19] [20] [24] [46] Next.js vs Nuxt vs SvelteKit: Choosing the Right Framework for SaaS Development | supastarter - SaaS starter kit for Next.js, Nuxt and SvelteKit
https://supastarter.dev/blog/nextjs-vs-nuxt-vs-sveltekit-for-saas-development

[21] [22] [23] [25] [48] Nuxt Explained: The Vue.js Framework Guide for 2025
https://strapi.io/blog/nuxt-vue-framework-explained-guide

[26] [27] [28] [30] [31] [37] [38] [40] Comparing Next.js, Astro, and Remix: Which is the Best?
https://strapi.io/blog/nextjs-vs-astro-vs-remix-choosing-the-right-frontend-framework

[29] Astro vs Next.js: Which Is Better for Your Project in 2025? - Pagepro
https://pagepro.co/blog/astro-nextjs/

[39] Next.js, Remix, and Astro - Which is Right for your Business?
https://bejamas.com/blog/next-js-remix-and-astro-which-is-right-for-your-business

[49] [50] [53] [54] [55] [87] [88] Node.js vs FastAPI for Building APIs - PLANEKS
https://www.planeks.net/nodejs-vs-fastapi-for-api/

51 62 67 68 Creating a Scalable Full-Stack Web App with Next.js and FastAPI | by Vijay Potta (pottavijay) | Medium
https://medium.com/@pottavijay/creating-a-scalable-full-stack-web-app-with-next-js-and-fastapi-eb4db44f4f4e

52 57 58 59 Comparing Three Favourite AI Technologies: Go, Node.js, and Python | by Darren Hinde | Medium
https://medium.com/@999daza/comparing-three-favourite-ai-technologies-go-node-js-and-python-76b305e8f372

56 60 61 85 86 Bench-marking RESTful APIs
https://gochronicles.com/benchmark-restful-apis/

63 Implementing Server-Sent Events (SSE) with FastAPI: Real-Time ...
https://medium.com/@mahdijafaridev/implementing-server-sent-events-sse-with-fastapi-real-time-updates-made-simple-6492f8bfc154

64 Building a Server-Sent Events (SSE) MCP Server with FastAPI - Ragie
https://www.ragie.ai/blog/building-a-server-sent-events-sse-mcp-server-with-fastapi

65 How to Stream LLM Responses in Real-Time Using FastAPI and SSE
https://blog.gopenai.com/how-to-stream-llm-responses-in-real-time-using-fastapi-and-sse-d2a5a30f2928

66 What is correct way to send streaming response from a POST API?
https://www.reddit.com/r/FastAPI/comments/1bwfjpl/what_is_correct_way_to_send_streaming_response/

69 70 LanguageTool - Supported Languages
https://dev.languagetool.org/languages

71 Turkish - LanguageTool Forum
https://forum.languagetool.org/t/turkish/328

72 [PDF] Açık Kaynak Doğal Dil İşleme Kütüphaneleri - DergiPark
https://dergipark.org.tr/en/download/article-file/1573501

73 74 75 emo.org.tr
https://www.emo.org.tr/ekler/c7a625d5077d3ba_ek.pdf?dergi=4

76 language-tool-python - PyPI
https://pypi.org/project/language-tool-python/

77 yeniguno/mbart50-turkish-grammar-corrector - Hugging Face
https://huggingface.co/yeniguno/mbart50-turkish-grammar-corrector

78 79 Grammatical Error Correction and Detection Dataset for Turkish
https://huggingface.co/papers/2309.11346

80 81 mukayese/mbart-large-turkish-summarization · Hugging Face
https://huggingface.co/mukayese/mbart-large-turkish-summarization

82 ozcangundes/mt5-small-turkish-summarization - Hugging Face
https://huggingface.co/ozcangundes/mt5-small-turkish-summarization

83 84 5 Farklı Açık-Kaynak Dil Modelinin Türkçeleri | Medium
https://medium.com/@ktoprakucar/5-farkl%C4%B1-a%C3%A7%C4%B1k-kaynak-dil-modelinin-t%C3%BCrk%C3%A7e-yetkinliklerinin-kar%C5%9F%C4%B1la%C5%9Ft%C4%B1r%C4%B1lmas%C4%B1-5c763749aa7e