

Programlama Dillerinde Bellek Güvenliği ve Ağ İletişimi: Rust, Go, Haskell ve Diğer Modern Dillerin Siber Güvenlik Odaklı Karşılaştırmalı ve Ampirik Analizi

Memory Safety and Network Communication in Programming Languages: A Cyber Security-Focused Comparative and Empirical Analysis of Rust, Go, Haskell and Other Modern Languages

Merve YILDIRIM¹, Muhammed SEYREK^{2*}

Öz

Bankacılık, telekomünikasyon (5G/6G), savunma sanayii ve nesnelerin interneti gibi kritik altyapılar büyük ölçüde yazılıma bağımlı hâle gelmiştir; bu yazılımların güvenilirliği ise bellek güvenliği ve eşzamanlılık doğruluğu gibi düşük seviyeli garantilerle doğrudan ilişkilidir. Bu çalışma, ağ iletişimi ve siber güvenlik bağlamında kullanılan programlama dillerini bellek güvenliği modeli, performans, ekosistem olgunluğu ve geliştirme ekonomisi açısından karşılaştırmaktadır. Çalışma üç bağımsız veri kaynağına dayanmaktadır: on bir dilde yürütülen kontrollü mikro-ölçüm deneyi, büyük ölçekli bir geliştirici anketinin (n = 49.191) ham mikro-verisi ve kurumsal geçiş literatürü. Ölçümler, Rust'ın sahiplik modelinin tahsis yoğun iş yüklerinde Go'nun çöp toplayıcısından yaklaşık 1,8 kat daha hızlı olduğunu, makine koduna derlenen dillerin soğuk başlatmada JVM'e kıyasla yaklaşık 16 kat daha hızlı olduğunu ve ağ katmanında Rust'ın hem gecikme hem verim bakımından C ile aynı düzeyde kaldığını göstermiştir. Anket analizi, Rust'ın %72,5 ile en yüksek sadakat oranına sahip olmasına karşın fiili kullanımının %9,6'da kaldığını ortaya koymuştur. Bulgular, tek bir dilin tüm senaryolar için en uygun seçenek olmadığını, bileşen sorumluluğuna göre seçilen karma (polyglot) mimarinin en dengeli sonucu verdiğini göstermektedir. Türkiye alt örnekleminde C'nin kullanım oranı küresel ortalamanın 1,65 katı iken Rust'ın oranı küresel düzeyde kalmaktadır. Çalışma, Türkiye için kademeli bir geçiş yol haritası önermektedir.

Anahtar Kelimeler: Bellek güvenliği, Ağ programlama, Rust, Ampirik ölçüm, Siber güvenlik, Polyglot mimari.

Abstract

Critical infrastructures such as banking, telecommunications (5G/6G), defence and the Internet of Things depend heavily on software, whose reliability is directly tied to low-level guarantees such as memory safety and concurrency correctness. This study compares programming languages used in network communication and cybersecurity in terms of memory safety model, performance, ecosystem maturity and development economics. It draws on three independent evidence sources: a controlled micro-benchmark experiment covering eleven languages, raw microdata from a large developer survey (n = 49,191), and the literature on institutional migration. The measurements show that Rust's ownership model is roughly 1.8 times faster than Go's garbage collector under allocation-heavy workloads, that natively compiled languages start roughly 16 times faster from cold than the JVM, and that Rust matches C in both network latency and throughput. Rust has the highest loyalty rate at 72.5%, while actual usage remains at 9.6%. No single language is optimal for every scenario; a polyglot architecture, with languages chosen by component responsibility, gives the most balanced outcome. In the Turkish subsample, C usage is 1.65 times the global average while Rust remains at the global level. The study proposes a phased migration roadmap for Türkiye.

Keywords: Memory safety, Network programming, Rust, Empirical measurement, Cyber security, Polyglot architecture.

¹Karadeniz Teknik Üniversitesi, Yazılım Mühendisliği Bölümü, Trabzon, Türkiye

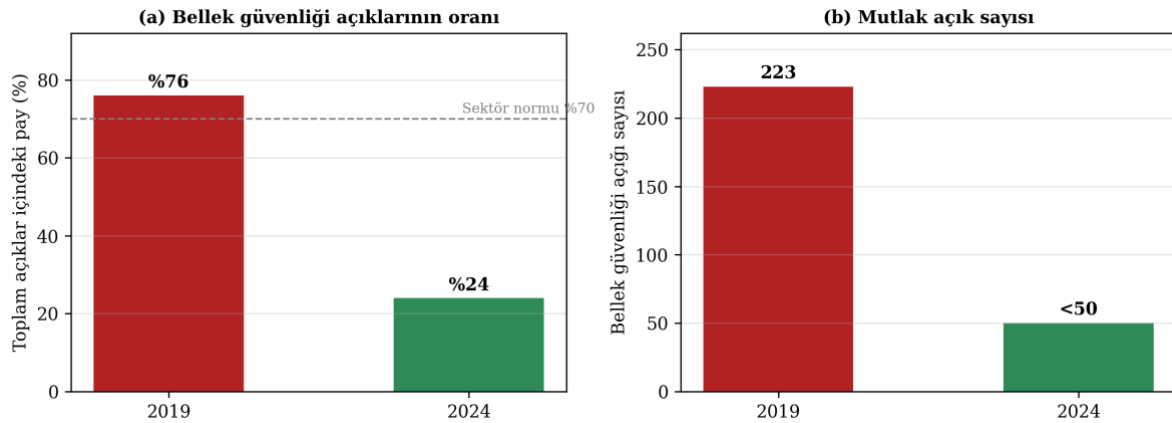
²Karadeniz Teknik Üniversitesi, Yapay Zeka ve Veri Mühendisliği Bölümü, Trabzon, Türkiye

*Sorumlu Yazar/Corresponding Author: muhammedseyrek00@gmail.com

1. Giriş

Modern bilişim altyapısının büyük bölümü, on yıllar önce tasarlanmış programlama dilleri üzerine inşa edilmiştir. Bu dillerin sunduğu yüksek performans, çoğu zaman bellek yönetiminin programcının sorumluluğuna bırakılması pahasına elde edilmiştir. Heartbleed (OpenSSL'de sınır dışı okuma, CVE-2014-0160) ve WannaCry (EternalBlue/SMBv1 bellek bozulmasından yararlanan uzaktan kod yürütme zinciri) gibi büyük çaplı siber olaylar, saldırı yüzeyinin önemli bir bölümünün doğrudan bellek hatalarıyla ilişkili olduğunu göstermiştir. SolarWinds saldırısı ise bellek güvenliğinden bağımsız biçimde, derleme ve dağıtım sürecinin hedef alınmasıyla gerçekleşmiş; dil seçiminin güvenliğin yalnızca bir boyutu olduğunu ortaya koymuştur.

Bellek güvenliği açıkları en yaygın güvenlik açığı sınıflarından biridir; MITRE'nin 2024 CWE Top 25 listesinde sınır dışı yazma (out-of-bounds write, CWE-787) ikinci sırada yer almaktadır (MITRE, 2024). Bellek güvenli dillere geçişin ölçülebilir bir etki ürettiği saha verileriyle desteklenmektedir: Google, Android'de yeni kodu bellek güvenli dillere kaydırarak bellek kaynaklı açıkların toplam içindeki payını 2019'daki %76 düzeyinden 2024'te %24'e indirmiş; mutlak sayı bakımından ise bu açıklar 223'ten 50'nin altına gerilemiştir (Vander Stoep ve Rebert, 2024). Söz konusu değişim Şekil 1'de gösterilmektedir. Tampon taşması (buffer overflow), serbest bırakılana erişim (use-after-free) ve veri yarışı (data race) bu sınıfın en sık görülen temsilcileridir. Bu açıkların ortak özelliği, çoğunlukla çalışma zamanında ve genellikle bir saldırgan tarafından bilinçli olarak tetiklendiğinde ortaya çıkmalarıdır; bu da onları tespit edilmesi en zor hata türlerinden biri yapar.



Şekil 1. Android platformunda bellek güvenliği açıklarının 2019–2024 arasındaki değişimi

Bu eğilim, ulusal siber güvenlik kurumlarını da harekete geçirmiştir. ABD Siber Güvenlik ve Altyapı Güvenliği Ajansı ile Ulusal Güvenlik Ajansı, kurumları bellek güvenli dillere geçiş yol haritaları hazırlamaya çağırmış (CISA vd., 2023; NSA, 2025); Beyaz Saray Ulusal Siber Direktörlük

Ofisi de aynı yönde bir politika belgesi yayımlamıştır (The White House, 2024). DARPA ise mevcut C kod tabanlarının Rust'a otomatik çevrimini hedefleyen TRACTOR programını başlatmıştır (DARPA, 2024). Düşük yazılım kalitesinin yalnızca Amerika Birleşik Devletleri ekonomisine maliyeti 2022 yılı için 2,41 trilyon dolar olarak tahmin edilmektedir (CISQ, 2022).

Bellek güvenli dillere geçişin önündeki en yaygın gerekçe, güvenlik garantilerinin performans pahasına elde edildiği varsayımdır. Bu varsayım sıklıkla dile getirilmekle birlikte, farklı bellek yönetimi modellerinin maliyetini aynı iş yükü üzerinde karşılaştıran ölçümlere sınırlı ölçüde dayanmaktadır. Bu çalışma, söz konusu maliyeti nicel olarak belirlemeyi ve dil seçiminin güvenlik sonuçlarını karar vericiler için kanıta dayalı bir çerçeveye oturtmayı amaçlamaktadır. Bu amaçla dil seçiminin güvenlik ve başarımlar sonuçları, kontrollü bir ölçüm deneyi ile büyük ölçekli bir geliştirici anketinin ham verisi üzerinde yürütülen analiz birlikte değerlendirilmektedir.

1.1. Araştırma Soruları

Çalışma aşağıdaki dört araştırma sorusuna yanıt aramaktadır:

AS1. Bellek güvenliği garantisi sağlayan diller, bu garantiyi ölçülebilir bir performans maliyetiyle mi elde etmektedir? Özellikle derleme zamanı denetimine dayanan sahiplik modeli ile çalışma zamanı çöp toplama arasındaki maliyet farkı nedir?

AS2. Yürütme modeli (makine koduna derleme, JIT, yorumlama) soğuk başlatma süresini ne ölçüde belirlemektedir ve bu fark hangi dağıtım senaryolarında anlamlı hâle gelmektedir?

AS3. Dillerin uygulama alanlarına göre fiili dağılımı, ölçülen teknik özelliklerle örtüşmekte midir?

AS4. Teknik üstünlük ve geliştirici memnuniyeti, bir dilin benimsenmesini açıklamakta yeterli midir?

1.2. Çalışmanın Özgün Katkısı

Bu çalışma literatüre üç noktada katkı sunmaktadır. Birincisi, on bir programlama dilinde aynı algoritmanın uygulandığı ve doğruluğu ortak bir sağlama değeriyle denetlenen bir mikro-ölçüm deneyi yürütülmüş; böylece bellek güvenliği ile performans arasındaki ödünleşim doğrudan nicelleştirilmiştir. İkincisi, büyük ölçekli bir geliştirici anketinin ham mikro-verisi üzerinde uygulama alanı $\times$ dil çapraz tabloları üretilmiş; böylece dillerin alanlara göre dağılımı yayımlanmış özet raporlar yerine yanıtlayıcı düzeyindeki kayıtlardan hesaplanmıştır. Üçüncüsü, ölçüm, literatür ve anket bulguları ayrı başlıklar altında sunulmuş; hakem denetiminden geçmemiş kurumsal raporların bulguları kendi ölçüm sonuçlarından ayrı tutulmuştur.

1.3. Temel Kavramlar

Bellek güvenliği (memory safety): Bir programın, kendisine ayrılmamış ya da artık geçerli olmayan bellek bölgelerine erişmesini engelleyen garantiler bütünüdür.

Tampon taşması (buffer overflow): Bir verinin, kendisine ayrılan tampon sınırlarının dışına yazılması; bitişik belleğin bozulmasına ve sıklıkla uzaktan kod yürütmeye yol açabilir.

Serbest bırakılana erişim (use-after-free): Belleğin serbest bırakılmasından sonra ona işaret eden bir göstericinin kullanılması; tanımsız davranış ve istismar edilebilir durumlar üretir.

Veri yarışı (data race): Birden fazla iş parçacığının senkronizasyon olmaksızın aynı bellek bölgesine eş zamanlı eriştiği ve bu erişimlerden en az birinin yazma olduğu durumdur; tanımsız davranışa ve öngörülemeyen sonuçlara yol açar.

Sahiplik (ownership): Her değerin tek bir sahibi olduğu ve sahip kapsam dışına çıktığında belleğin derleyici tarafından otomatik olarak serbest bırakıldığı bellek yönetim modelidir. Model Rust'ta en olgun biçimde uygulanmakla birlikte, kökeni C++'ın RAII yaklaşımına dayanmaktadır.

Güvensiz blok (unsafe): Rust'ta derleyicinin bellek güvenliği denetimlerinin programcı tarafından açıkça devre dışı bırakıldığı kod bölgesi; düşük seviyeli donanım erişimi ve yabancı dil arayüzleri için gereklidir.

Soğuk başlatma (cold start): Bir programın sıfırdan başlatılıp ilk faydalı işi yapmaya başlamasına kadar geçen süre; çalışma zamanı kurulumu, kütüphane yükleme ve JIT ısınması bu süreye dâhildir.

2. İlgili Çalışmalar

Bellek güvenliğinin ampirik olarak incelenmesi, son yıllarda yazılım mühendisliği yazınında belirgin bir araştırma hattı oluşturmuştur. Bu hat üç ekseninde gelişmiştir: tek bir dilin güvenlik garantilerini sınanan derinlemesine analizler, diller arası nicel karşılaştırmalar ve kurumsal geçiş deneyimleri.

2.1. Rust'ın Güvenlik Garantilerinin Ampirik İncelenmesi

Qin vd. (2020), beş açık kaynak Rust projesi, beş yaygın kütüphane ve iki güvenlik veri tabanı üzerinde 850 unsafe kod kullanımını ve 170 hatayı elle inceleyerek alanın ilk kapsamlı ampirik çalışmasını gerçekleştirmiştir. Çalışmanın en dikkate değer bulgusu, incelenen bellek güvenliği hatalarının tamamının unsafe kod kullanımıyla ilişkili olmasıdır; bu sonuç, güvenli Rust alt kümesinin sağladığı garantinin pratikte de geçerli olduğunu göstermektedir. Xu vd. (2021), 186 Rust CVE kaydı

üzerinde yürüttükleri incelemeyle bu sonucu bağımsız olarak doğrulamış ve inceledikleri kayıtlarda tanımsız davranışın kaynağının tümüyle unsafe kod olduğunu raporlamıştır.

Evans vd. (2020) ile Astrauskas vd. (2020) ise unsafe kullanımının yaygınlığını büyük ölçekli otomatik analizlerle incelemiş; unsafe bloklarının kod tabanlarında sınırlı bir paya sahip olduğunu ve çoğunlukla küçük, yalıtılmış bölgelerde yoğunlaştığını ortaya koymuştur. Ekosistem ölçeğinde ise Zheng vd. (2024), crates.io deposundaki paketlerde açıklanan on yedi açık türünü incelemiş; bellek güvenliği ve eşzamanlılık sorunlarının sınıflandırılan açıkların yaklaşık üçte ikisini oluşturduğunu ve bir açığın koda girmesinden kamuya açıklanmasına kadar ortalama 770 gün geçtiğini raporlamıştır.

İşletim sistemi çekirdeği gibi en zorlu bağlamlarda ise tablo daha karmaşıktır. Li vd. (2024), Rust-for-Linux projesinde altı sürücüyü, yüzlerce sorun kaydını ve on iki binin üzerinde geliştirici tartışmasını inceleyerek Rust'ın çekirdek açıklarını azalttığını, ancak bunları bütünüyle ortadan kaldıramadığını göstermiştir. Yazarlar ayrıca, güvenlik güvencesinin uygun biçimde tasarlanmaması hâlinde hem çalışma zamanı yükü hem de geliştirme eforu bakımından belirgin bir maliyet doğurabildiğini raporlamıştır. Bu bulgu, bellek güvenli dillerin tek başına yeterli olmadığı ve Bölüm 6'da ele alınan sınırlılıkların ampirik dayanağını oluşturmaktadır.

Söz konusu çalışmaların tamamı tek bir dile odaklanmakta ve diller arası karşılaştırma yapmamaktadır.

2.2. Diller Arası Ampirik Karşılaştırma Çalışmaları

Diller arası ampirik karşılaştırmanın kökeni Prechelt (2000) çalışmasına dayanır; yazar aynı programlama görevini yedi dilde uygulayan yüzden fazla çözümü inceleyerek geliştirme eforu ile çalışma zamanı başarımı arasındaki ödünleşimi ortaya koymuştur. Nanz ve Furia (2015) bu çizgiyi büyük ölçeğe taşımış; Rosetta Code deposundaki 745 göreve ait 7.087 çözümü sekiz dilde (C, Go, C#, Java, F#, Haskell, Python, Ruby) karşılaştırarak ham hız söz konusu olduğunda C'nin üstünlüğünü istatistiksel olarak göstermiştir. Bu sonuç, mevcut çalışmada C ve C++'ın referans hızı belirlemesiyle örtüşmektedir (Bölüm 4.2).

Mevcut çalışmanın ölçüm tasarımı bakımından başlıca metodolojik referansı Pereira vd. (2017) çalışmasıdır. Yazarlar, Computer Language Benchmarks Game üzerinde on farklı problemi yirmi yedi dilde çalıştırarak çalışma süresi, bellek kullanımı ve enerji tüketimini birlikte ölçmüş; her ölçümü on kez tekrarlayarak aykırı değerleri elemiş ve medyan üzerinden raporlamıştır. Aynı ekibin genişletilmiş çalışması (Pereira vd., 2021) bu sonuçları doğrulamıştır. Çalışmanın en önemli çıkarımı, evrensel olarak üstün bir dilin bulunmadığı ve dil seçiminin kullanım bağlamına göre değiştiğidir.

Güvenlik ve başarımı birlikte ele alan sınırlı sayıdaki çalışmadan biri Bugden ve Alahmar (2022) tarafından gerçekleştirilmiştir. Yazarlar altı dili (C, C++, Go, Java, Python, Rust) hem nicel

ölçümler hem de nitel değerlendirmeye karşılaştırmış; Rust'ı güvenlik bakımından en üstün dil olarak belirlemiş ve başarımlar açısından Go, Java ve Python'un üzerinde, C ile C++'a yakın bir konumda bulunduğunu raporlamıştır. Mevcut çalışma, bu iki eksenli yaklaşımı benimsemekle birlikte üç noktada ayrılmaktadır: dil kapsamı on bire genişletilmiş, iş yükleri bellek yönetimi stratejisini doğrudan hedefleyecek biçimde seçilmiş ve ölçümler geliştirici benimseme verisiyle birlikte yorumlanmıştır.

Ölçüm tasarımı Pereira vd.'nin yöntemsel yaklaşımını izlemektedir: aynı algoritmanın diller arası eşdeğer uygulaması, çoklu tekrar ve medyan raporlama.

2.3. Kurumsal Geçiş Deneyimleri

Üçüncü eksen, bellek güvenli dillere geçişin saha sonuçlarını belgeleyen kurumsal raporlardır. Google'ın Android platformundaki geçiş deneyimi (Vander Stoep ve Rebert, 2024) ile ISRG'nin TLS kütüphanesi ölçümleri (Prossimo, 2025) bu grubun en sık atıf alan örnekleridir. Bu kaynaklar geniş ölçekli ve gerçek üretim verisine dayanmaları bakımından değerli olmakla birlikte, hakem denetiminden geçmemekte ve yöntemleri bağımsız olarak doğrulanamamaktadır.

2.4. Literatürdeki Boşluk

Mevcut yazın iki uçta yoğunlaşmaktadır: tek dile odaklanan derinlemesine güvenlik analizleri ile bağlamdan bağımsız genel performans sıralamaları. Bellek güvenliği tartışmasını doğrudan hedefleyen bir ölçüm tasarımını, dillerin uygulama alanlarına göre fiili dağılımıyla birlikte ele alan çalışma sayısı sınırlıdır. Ayrıca yapılan tarama kapsamında, Türkçe yazında kritik altyapı bağlamında dil seçimini ampirik verilere dayandıran bir çalışmaya rastlanmamıştır. Bu çalışma, söz konusu boşluğu üç bağımsız veri kaynağını birlikte kullanarak doldurmayı amaçlamaktadır.

3. Materyal ve Metot

Bu çalışmada üç ayrı veri kaynağı kullanılmış ve bulgular kaynak düzeyine göre ayrı başlıklar altında sunulmuştur. Aşağıda her kaynağın materyali ve yöntemi ayrı ayrı açıklanmaktadır.

3.1. Ampirik Ölçüm Deneyi

On bir programlama dili — C, C++, Rust, C#, Go, Java, Haskell, JavaScript (Node.js), PHP, Ruby ve Python — beş ölçülebilir eksenle karşılaştırılmıştır: hesaplama süresi, tahsis maliyeti, soğuk

başlatma süresi, ikili boyut ve ağ gecikmesi/verimi. Diller, farklı bellek yönetimi ve yürütme modellerini (manuel, sahiplik tabanlı, referans sayma, izleyici çöp toplama; makine koduna derlenen, JIT, yorumlanan) temsil edecek biçimde seçilmiştir.

İki iş yükü tasarlanmıştır. Birincisi (buffer_scan), 16 MB'lık bir tampon üzerinde on tur boyunca bit döndürme ve XOR işlemi uygulayarak saf hesaplama süresini, dal tahminini ve sınır denetimi maliyetini izole eder. İkincisi (alloc_churn), bir milyon değişken boyutlu tahsis gerçekleştirerek bellek yönetimi stratejisinin maliyetini ölçer. Karşılaştırmanın adilliğini güvence altına almak için tüm uygulamalar algoritmik olarak eşdeğer tutulmuş ve doğruluk denetimi olarak hepsinin aynı sağlama değerini üretmesi aranmıştır (21.390.950.400). Söz konusu denetim, dillerden hiçbirinin eksik ya da farklı iş yaparak avantaj kazanmadığını göstermektedir.

Ölçümler, Intel Core i7-13650HX işlemciye (20 mantıksal çekirdek) sahip bir sistemde, WSL2 üzerinde çalışan Ubuntu 24.04.5 (çekirdek 6.18) ortamında gerçekleştirilmiştir. Söz konusu işlemci hibrit mimariye sahip olduğundan ve sürecin performans ya da verimlilik çekirdeğinde çalışması ölçümler arasında farklılık yaratabileceğinden, tüm ölçümler taskset aracıyla tek bir çekirdeğe (0 numaralı çekirdek) sabitlenmiştir. Sisteme ayrılan bellek 15 GiB'dir. Ortamın tam donanım ve sürüm bilgisi çalışmanın ekinde (ORTAM.txt) sunulmaktadır.

Kullanılan sürümler şunlardır: GCC 13.3, Rust 1.75, .NET 8.0, Go 1.22, GHC 9.4, OpenJDK 21, Node.js 22, PHP 8.3, Ruby 3.2 ve Python 3.12. Her hesaplama ve tahsis ölçümü, ilk çalıştırma önbellek ve JIT ısınması olarak atıldıktan sonra tüm dillerde yirmi kez tekrarlanmış; merkezi eğilim ölçüsü olarak medyan raporlanmıştır. Ölçümlerin değişkenliği iki ayrı göstergeyle nicelleştirilmiştir: standart sapmaya dayalı değişim katsayısı (CV) ve medyan mutlak sapmaya dayalı dayanıklı değişim katsayısı ($rCV = MAD / \text{medyan}$). Standart sapma tek bir aykırı ölçümden belirgin biçimde etkilendiğinden, raporlamada rCV esas alınmış, CV ise karşılaştırma amacıyla birlikte verilmiştir. Medyandan beş MAD'den fazla sapan turlar aykırı olarak işaretlenmiş, ancak hesaplamadan çıkarılmamıştır; aykırı turların listesi ektteki istatistik dosyalarında yer almaktadır.

3.2. Anket Verisi Üzerinden İkincil Veri Analizi

İkinci veri kaynağı, kamuya açık bir geliştirici anketinin ham mikro-verisine dayanmaktadır. Stack Overflow Developer Survey 2025 veri seti 49.191 geçerli yanıt içermektedir; diller çoklu-seçim biçiminde toplandığından yüzdeler örneklem büyüklüğüne oranlanmıştır ve toplamaları %100'ü aşar. Oranlar, yayımlanmış özet raporlar yerine yanıtlayıcı düzeyindeki ham kayıtlardan hesaplanmıştır (Stack Overflow, 2025).

Analizde üç gösterge kullanılmıştır. Kullanım oranı, ilgili dili son bir yıl içinde kullandığını bildirenlerin tüm örnekleme oranıdır. Sadakat oranı, bir dili kullananlar içinde o dile hayranlık

duyduğunu bildirenlerin payıdır (hayran ÷ kullanan) ve dilin kullanıcılarını elde tutma gücünü ölçer. Net talep ise dili kullanmak isteyenler ile hâlihazırda kullananlar arasındaki farktır; pozitif değer dilin mevcut kullanıcı tabanından daha geniş bir ilgi gördüğünü, negatif değer ise dilin büyük ölçüde zorunluluktan kullanıldığını gösterir. Göstergelerin ayrıntılı tanımları Ek B.2'de verilmektedir. Yöntemin sınırlılıkları Bölüm 6'da ele alınmaktadır.

3.3. Kurumsal Geçiş Literatürü

Üçüncü veri kaynağı, bellek güvenli dillere geçişin saha sonuçlarını belgeleyen kurumsal raporlardan oluşmaktadır. Bu gruba, gerçek üretim ortamlarından elde edilen ve veri kaynağı açıkça belirtilen belgeler dâhil edilmiştir. Söz konusu kaynaklar hakem denetiminden geçemediğinden ve yöntemleri bağımsız olarak doğrulanamadığından, bulguları ölçüm sonuçlarıyla aynı düzeyde değerlendirilmemiş; Bölüm 4.9'da ayrı bir başlık altında sunulmuştur.

4. Bulgular ve Tartışma

4.1. Bellek Yönetimi Yaklaşımlarının Kavramsal Karşılaştırması

Diller bellek güvenliğini farklı stratejilerle ele almakta ve her strateji, güvenlik garantisi ile çalışma zamanı maliyeti arasında farklı bir denge kurmaktadır. Manuel yönetimde (C/C++) programcı belleği elle ayırır ve serbest bırakır; bu en yüksek denetimi sağlar ancak use-after-free, çift serbest bırakma ve sızıntı gibi hatalara açıktır. Otomatik yönetimde sorumluluk çalışma zamanına devredilir ve güvenlik büyük ölçüde sağlanır. Bu başlık altında iki ayrı mekanizma yer alır: izleyici çöp toplama (Go, Java, C#, Haskell) belleği toplu hâlde geri kazanır ve öngörülemez duraklamalar ile ek bellek ayak izi getirebilir; referans sayma (Python, PHP) ise maliyeti programın geneline yayar, ancak döngüsel referansları tek başına çözemediği için ek bir toplayıcı gerektirir. Rust'ın sahiplik modeli ise belleği derleme zamanında denetler: ne manuel yönetimin riskini ne de çöp toplamanın çalışma zamanı maliyetini taşır. Bu ayrımın performans sonuçları izleyen alt bölümlerde sayısal olarak sınanmaktadır.

4.2. Hesaplama Süresi

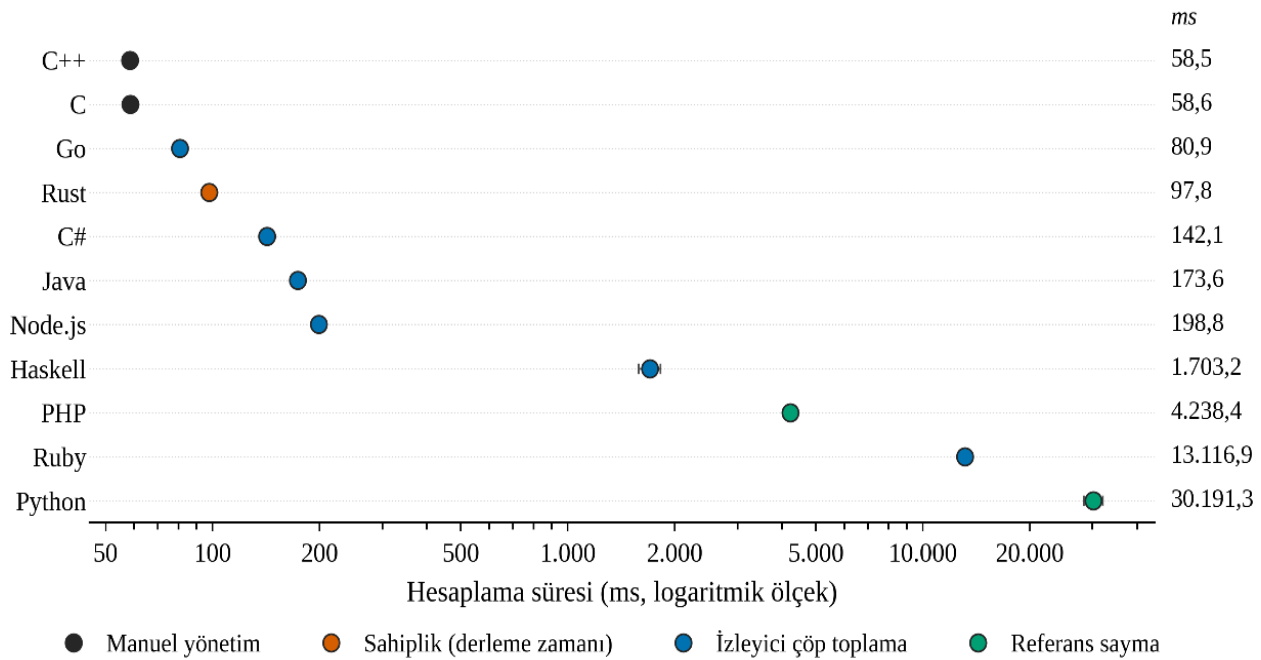
Şekil 2'de gösterildiği üzere, hesaplama ekseninde C ve C++ referans hızı belirlemektedir (58,5–58,6 ms). Go 80,9 ms ile bunların yaklaşık 1,38 katı içinde kalmakta, Rust ise 97,8 ms ile 1,67 katında yer almaktadır. Rust ile C arasındaki bu fark büyük ölçüde Rust'ın varsayılan olarak

uyguladığı dizi sınır denetimlerinden kaynaklanmaktadır. Bu gözlem, Rust'ı C ile aynı iş yükünde karşılaştıran ve yavaşlığın temel kaynağını derleyicinin eklediği çalışma zamanı denetimlerine bağlayan bağımsız ölçüm çalışmalarıyla tutarlıdır (Zhang vd., 2022). Dolayısıyla güvenliğin hesaplama ekseninde ölçülebilir bir bedeli vardır; bu bedel, karşılığında sağlanan garanti göz önüne alındığında sınırlı sayılabilir.

JIT derlemesi kullanan çalışma zamanları ara bir konumdadır: C# (.NET 8) 142,1 ms, Java 173,6 ms ve Node.js (V8) 198,8 ms ile birbirine yakın değerler vermiştir. Bu üç çalışma zamanının makine koduna derlenen dillere oranı 2,4 ilâ 3,4 kat aralığındadır.

Yorumlanan diller ile tembel değerlendirmeye dayanan Haskell ise belirgin biçimde geride kalmaktadır (Haskell 1,7 s; PHP 4,2 s; Ruby 13,1 s; Python 30,2 s). Haskell'in bu ekseninde zayıf kalması, uygulamada kullanılan endeksli ByteString erişiminin dilin deyimsel kullanımını yansıtmamasıyla ilgilidir; bu sınırlılık Bölüm 6'da ele alınmaktadır. PHP'nin Ruby ve Python'dan sırasıyla üç ve yedi kat hızlı olması, çalışma zamanı mimarisinin dil sınıfından daha belirleyici olabileceğini göstermektedir.

Hesaplama ölçümlerinin dayanıklı değişim katsayısı Haskell (%7,1) ve Python (%6,2) dışında tüm dillerde %2'nin altında kalmıştır (Tablo 1). Standart sapmaya dayalı CV değerleri ise bazı dillerde belirgin biçimde yüksektir (Haskell %42,3; PHP %28,1; C# %16,3); bu fark, her birinde bir ilâ üç adet aykırı turun bulunmasından kaynaklanmaktadır. Söz konusu turlar hesaplamadan çıkarılmamış, ekteki istatistik dosyalarında ayrıca listelenmiştir.



Şekil 2. On bir dilde hesaplama süresi.

4.3. Tahsis Maliyeti

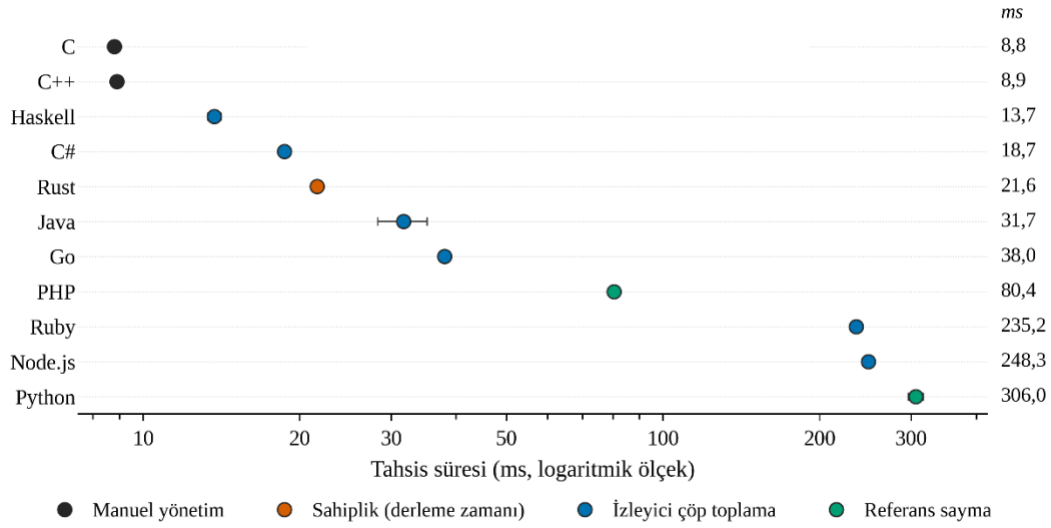
Tahsis yoğun iş yükünde en hızlı grubu, manuel bellek yönetimi uygulayan C ve C++ oluşturmaktadır (8,8 ve 8,9 ms). Bunları Haskell (13,7 ms), C# (18,7 ms) ve Rust (21,6 ms) izlemekte; Java 31,7 ms ve Go 38,0 ms ile daha geride yer almaktadır. Tahsis maliyeti sonuçları Şekil 3'te logaritmik ölçekte sunulmuş; noktalar medyanı, yatay çubuklar ise medyan mutlak sapmayı (MAD) göstermektedir.

Bu sıralama, bellek yönetimi stratejisi ile tahsis maliyeti arasındaki ilişkinin basit bir ikiliğe indirgenemeyeceğini ortaya koymaktadır. Derleme zamanı denetimine dayanan Rust, çöp toplayıcı kullanan Go'dan 1,76 ve Java'dan 1,47 kat hızlıdır; buna karşılık yine çöp toplayıcı kullanan Haskell ve C#, Rust'ın önünde yer almaktadır. Belirleyici olan, belleğin çöp toplayıcı tarafından yönetilip yönetilmediği değil, tahsis mekanizmasının kendisidir: kuşak tabanlı toplayıcılarda yeni nesneler, yalnızca bir işaretçinin ilerletilmesiyle ayrılan bir fidelik (nursery) alanında oluşturulur ve bu, kısa ömürlü nesnelerin yoğun olduğu bu tür iş yüklerinde çok düşük bir maliyet üretir. Go'nun boyut sınıfına dayalı ayırıcısı ise aynı iş yükünde en yüksek maliyeti göstermiştir.

Bununla birlikte, çöp toplamının maliyeti yalnızca ortalama süreyle ölçülemez. Java'nın tahsis ölçümlerinde dayanıklı değişim katsayısı %10,9'dur ve bu değer, incelenen on bir dilin tamamı içinde en yüksek olanıdır; diğer dillerde aynı gösterge %3,4'ün altında kalmaktadır. Ham ölçüm kayıtları, Java'nın tahsis sürelerinin 27,5 ile 43,1 ms arasında dalgalandığını göstermektedir. Söz konusu değişkenlik, Bölüm 4.1'de kavramsal olarak tartışılan öngörülemez duraklamaların doğrudan ölçülmüş karşılığıdır ve gecikme güvencesi gerektiren sistemlerde ortalama sürenin tek başına yeterli bir ölçüt olmadığını göstermektedir.

Yorumlanan ve sanal makine tabanlı dillerde tahsis maliyeti belirgin biçimde yükselmektedir (PHP 80,4 ms; Ruby 235,2 ms; Node.js 248,3 ms; Python 306,0 ms). Bu artış, referans sayma ve dinamik tip denetiminin her tahsise eklediği yükü yansıtmaktadır.

Bulgular bir bütün olarak değerlendirildiğinde, bellek güvenliğinin performansla zorunlu olarak çelişmediği görülmektedir: derleme zamanı denetimine dayanan Rust, manuel yönetimin yaklaşık 2,45 katı bir maliyetle güvenlik garantisi sunmakta ve bellek güvenli dillerin tamamıyla aynı büyüklük mertebesinde kalmaktadır.



Şekil 3. On bir programlama dilinde tahsis maliyeti.

4.4. Soğuk Başlatma Süresi

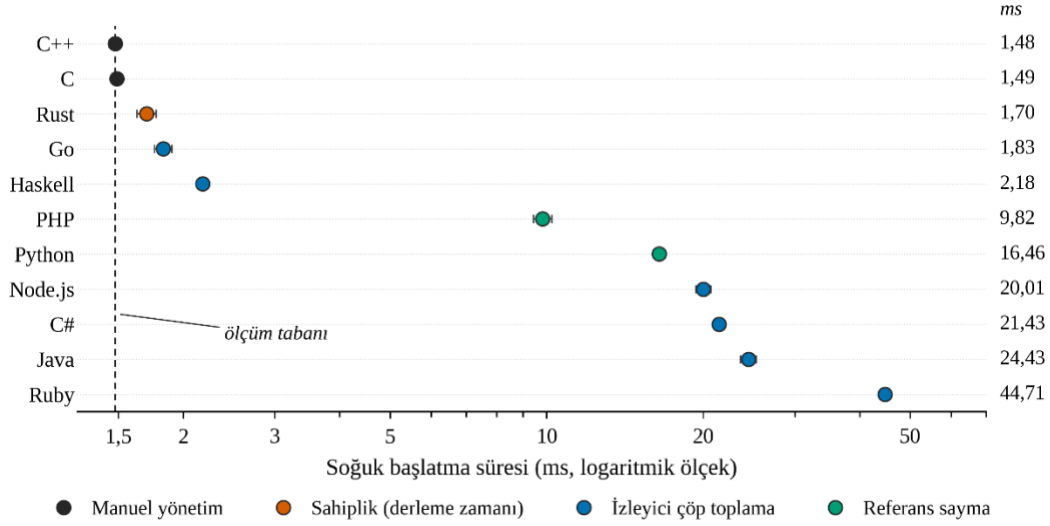
Başlatma süresi, uzun ömürlü sunucu süreçleri için ihmal edilebilirken; serverless (FaaS) fonksiyonları, komut satırı araçları ve sık yeniden başlatılan kenar (edge) hizmetleri için doğrudan belirleyicidir; bu senaryolarda her soğuk başlatma söz konusu gecikmeyi yeniden üretmektedir.

Ölçüm, her dilde yalnızca çıkış yapan boş bir programın yirmi kez çalıştırılmasıyla yapılmış; böylece yalnızca çalışma zamanı kurulumu, kütüphane yükleme ve JIT ilklandirmesi izole edilmiştir. Raporlanan değerler, ölçüm sarmalayıcısının 1,48 ms'lik sabit ek yükünü içermektedir; bu taban çizgisi tüm dillerde ortaktır ve ayrıca ölçülerek ekte kaydedilmiştir.

Makine koduna derlenen diller 2,2 ms'nin altında başlatılmaktadır (C++ 1,48 ms; C 1,49 ms; Rust 1,70 ms; Go 1,83 ms; Haskell 2,18 ms). Taban çizgisi çıkarıldığında bu dillerin net başlatma süresi 0,7 ms'nin altında kalmakta, C ve C++'ta ise ölçüm çözünürlüğünün altına inmektedir. Buna karşılık sanal makine ve yorumlayıcı tabanlı diller 9,8 ilâ 44,7 ms aralığına yayılmaktadır (PHP 9,82 ms; Python 16,46 ms; Node.js 20,01 ms; C# 21,43 ms; Java 24,43 ms; Ruby 44,71 ms). JVM ile C arasındaki yaklaşık on altı katlık fark, saniyede binlerce kısa ömürlü çağrı alan bir sistemde toplam gecikmeye doğrudan yansımaktadır. Bu bulgu, CDN ve kenar bilişim bileşenleri için dil seçiminin neden makine koduna derlenen dillere yöneldiğini açıklamaktadır. Soğuk başlatma sonuçları Şekil 4'te logaritmik ölçekte sunulmakta; kesikli çizgi tüm diller için ortak olan 1,48 ms'lik ölçüm tabanını göstermektedir.

İkili dosya boyutu ekseninde ise farklı bir sıralama ortaya çıkmaktadır: C ve C++ 14 KB ile en küçük yürütülebilir dosyaları üretirken, Rust 362 KB, Go 868 KB ve Haskell 5.529 KB ile belirgin biçimde daha büyük çıktılar vermektedir. Bu fark, söz konusu dillerin çalışma zamanlarını yürütülebilir dosyaya statik olarak bağlamasından kaynaklanmaktadır ve konteyner imajı boyutunun

önem taşıdığı dağıtım senaryolarında dikkate alınması gereken bir ödünleşim oluşturmaktadır. İkili boyutlar, sembolleri ayıklanmış (stripped) yürütülebilir dosyalar üzerinden belirlenmiştir; sanal makine ve yorumlayıcı tabanlı dillerde tek bir yürütülebilir dosya üretilmediğinden ilgili değerler raporlanmamıştır. Hesaplama, tahsis, soğuk başlatma ve ikili boyut ölçümlerinin tümü Tablo 1’de özetlenmiştir.



Şekil 4. On bir programlama dilinde soğuk başlatma süresi.

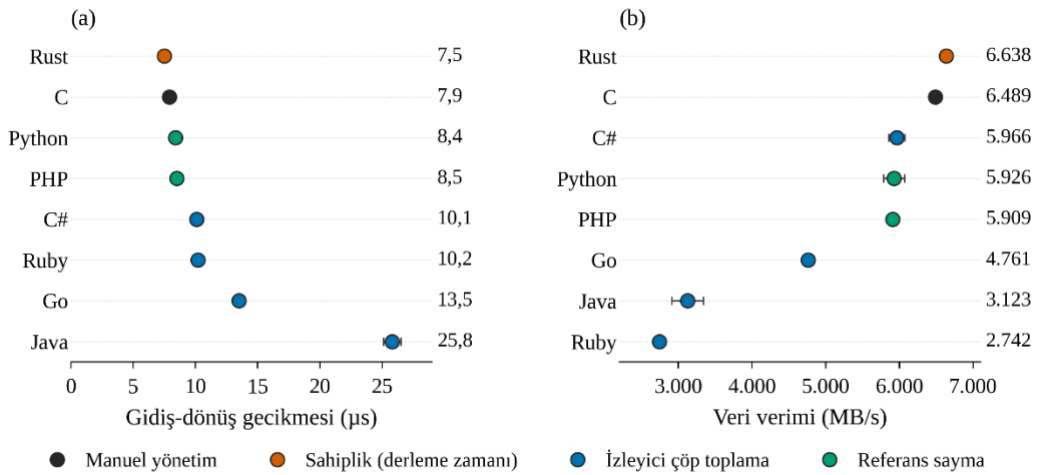
Tablo 1. On bir programlama dilinde performans ölçüm sonuçları.

Dil	Hesaplama (ms)	rCV (%)	Tahsis (ms)	rCV (%)	Başlatma (ms)	İkili (KB)
C++	58,5	0,7	8,9	1,7	1,48	14
C	58,6	1,1	8,8	0,0	1,49	14
Go	80,9	1,1	38,0	1,1	1,83	868
Rust	97,8	0,6	21,6	1,6	1,70	362
C#	142,1	0,9	18,7	2,4	21,43	—
Java	173,6	1,8	31,7	10,9	24,43	—
Node.js	198,8	0,9	248,3	1,2	20,01	—
Haskell	1.703,2	7,1	13,7	2,9	2,18	5.529
PHP	4.238,4	0,7	80,4	1,7	9,82	—
Ruby	13.116,9	0,5	235,2	2,0	44,71	—
Python	30.191,3	6,2	306,0	3,4	16,46	—

4.5. Ağ İletişimi Başarımı

Bu çalışmanın merkezî bağlamı ağ iletişimi olduğundan, sekiz dil (C, Rust, C#, Go, Java, Python, PHP ve Ruby) için gerçek bir TCP yankı (echo) ölçümü gerçekleştirilmiştir. Her dilde

birbirinin eşdeğeri bir sunucu ve istemci yazılmış; istemci önce 64 baytlık 10.000 gidiş-dönüş (ping-pong) işlemiyle ortalama tur süresi gecikmesini, ardından 256 MB'lık çift yönlü blok aktarımıyla veri verimini ölçmüştür. Gecikme ve verim ölçümleri sekiz dilde aynı protokol ve aynı blok boyutuyla yürütülmüştür. Ölçüm geri döngü (loopback) arayüzü üzerinde yapıldığından ağ kartı ve fiziksel bağlantı değişkenleri dışlanmış, yalnızca dilin ve çalışma zamanının soket işleme maliyeti izole edilmiştir. Her ölçüm yirmi kez tekrarlanmıştır. Ağ gecikmesi ve verim sonuçları Şekil 5'te gösterilmektedir.



Şekil 5. Programlama dillerinin ağ başarımı: (a) gidiş-dönüş gecikmesi, (b) çift yönlü veri verimi.

Buradaki en dikkate değer bulgu, Python ve PHP gibi yorumlanan dillerin ağ gecikmesinde Go ve Java'nın önüne geçmesidir. Bunun nedeni, bu dillerde soket işlemlerinin doğrudan işletim sisteminin C ile yazılmış çekirdek çağrılarına indirgenmesi; buna karşılık Go ve Java çalışma zamanlarının kendi zamanlayıcı ve tampon yönetim katmanlarını araya eklemesidir. Hesaplama-yoğun iş yüklerinde çok yavaş olan bu diller, saf soket iletişiminde işi büyük ölçüde çekirdeğe devrettikleri için rekabetçi kalabilmektedir. Bu ayrım, ağ hizmetinin darboğazının hesaplama mı yoksa saf G/Ç mü olduğuna göre dil seçiminin değişmesi gerektiğini göstermektedir.

Rust'ın hem gecikme hem verim ekseninde C ile aynı düzeyde yer alması, bellek güvenliği garantisinin yüksek başarılı ağ iletişimiyle çelişmediğini göstermektedir. Bu ölçümün geri döngü (loopback) arayüzü üzerinde yapıldığı ve gerçek ağ koşullarında fiziksel gecikmenin diller arası farkları görece küçülteceği, Bölüm 6'da bir sınırlılık olarak ele alınmaktadır. Ağ başarımına ilişkin sayısal sonuçlar Tablo 2'de özetlenmiştir. Ağ ölçümlerinin dayanıklı değişim katsayısı tüm dillerde %2,7'nin altında kalmıştır; buna karşılık birbirine yakın değer veren diller arasındaki sıralama kesin kabul edilmemeli, yalnızca büyüklük mertebesi farkları yorumlanmalıdır.

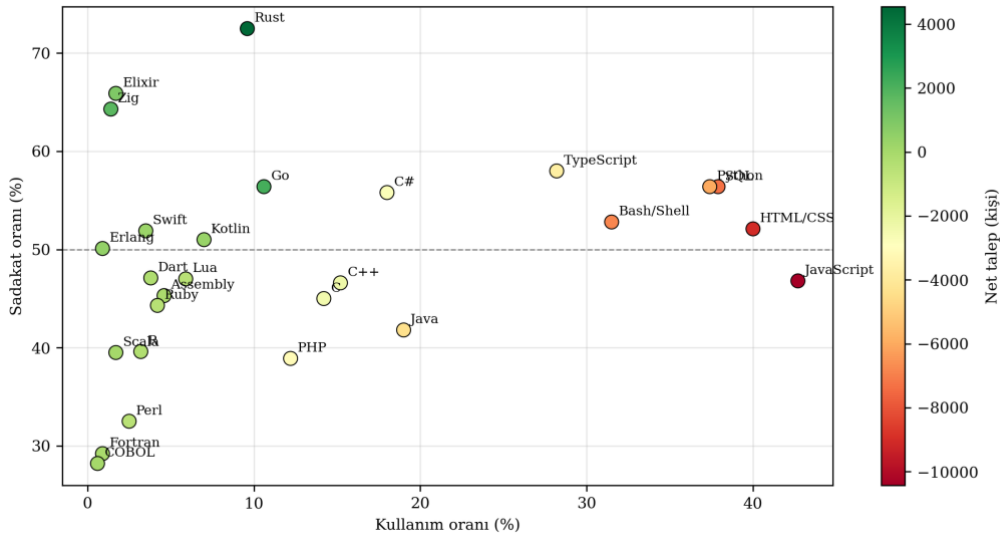
Java'nın verim ölçümünde dayanıklı değişim katsayısı %7,0'dır ve bu değer sekiz dil içinde en yüksek olanıdır; Bölüm 4.3'te tahsis ölçümünde gözlenen örüntü ağ katmanında da tekrarlanmaktadır.

Tablo 2. Programlama dillerinin ağ başarımı ölçüm sonuçları.

Dil	Gecikme (ms)	rCV (%)	Verim (MB/s)	rCV (%)
Rust	0,0075	2,7	6.637,8	0,8
C	0,0079	2,5	6.489,1	1,1
Python	0,0084	2,4	5.925,6	2,4
PHP	0,0085	2,4	5.909,2	1,0
C#	0,0101	2,0	5.965,6	1,9
Ruby	0,0102	1,0	2.742,1	1,1
Go	0,0135	2,6	4.761,0	1,1
Java	0,0258	2,7	3.123,0	7,0

4.6. Geliştirici Sadakati ve Benimseme Örüntüsü

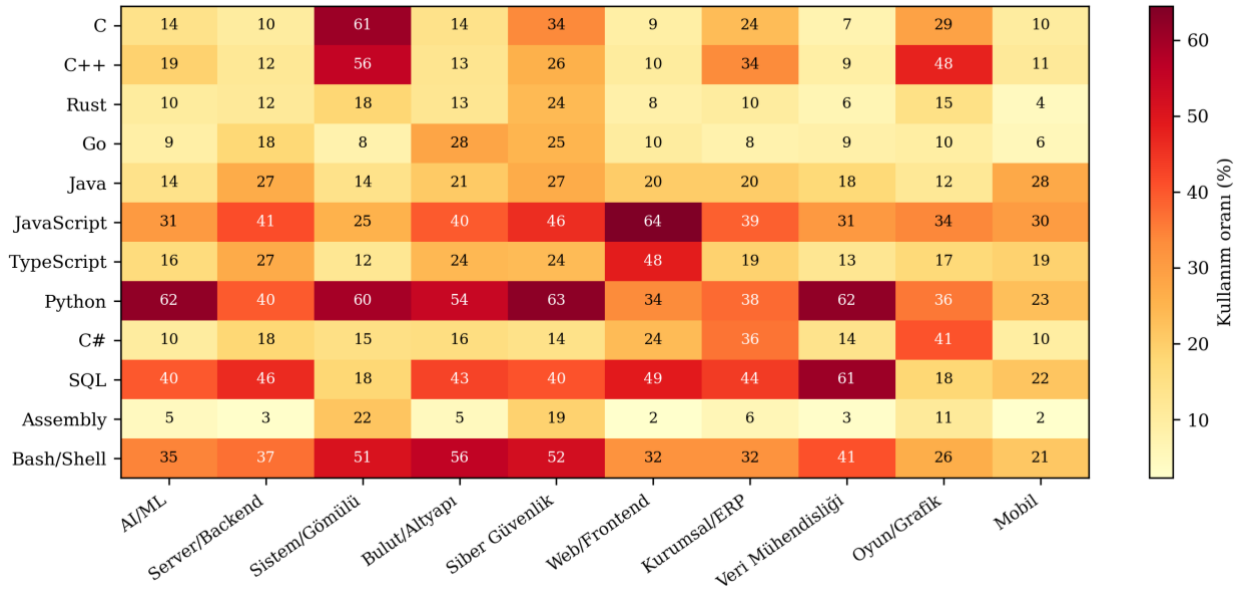
Anket verisi üç örüntü ortaya koymaktadır. Birincisi, yaygın diller ile sadık kitleye sahip diller aynı değildir: JavaScript %42,7 ile en yaygın dil olmasına karşın sadakat oranı %46,8'de kalmakta ve net talebi -10.424 ile tüm diller içinde en olumsuz değeri almaktadır. Bu durum, JavaScript'in büyük ölçüde web platformunun teknik zorunluluğundan kullanıldığına işaret etmektedir. İkincisi, Rust %72,5 sadakat oranıyla açık ara birincidir ve +4.538 (örneklemin yaklaşık %9,2'si) ile en yüksek net talebe sahiptir; buna karşın fiili kullanım oranı yalnızca %9,6'dır. Düşük kullanım, yüksek sadakat ve yüksek talebin bir arada bulunması, yaygınlaşma aşamasındaki bir dilin örüntüsüdür. Üçüncüsü, eski nesil dillerde sadakat belirgin biçimde düşüktür; COBOL %28,2 ve Fortran %29,2 ile en düşük oranlara sahiptir ve bu, bakım yükü altındaki eski sistemlerin sürdürülebilirlik riskine dair ölçülebilir bir göstergedir. Kullanım yaygınlığı ile geliştirici sadakati arasındaki ilişki Şekil 6'da gösterilmekte, nokta rengi ise net talebi temsil etmektedir.

**Şekil 6.** Kullanım yaygınlığı ile geliştirici sadakati arasındaki ilişki.

Bu bulgu, Bölüm 4.3'teki ölçümle birlikte okunduğunda anlam kazanmaktadır: Rust'ın ölçülen başarımı, geliştiricilerin öznel değerlendirmesiyle aynı yöne işaret etmektedir. Ancak memnuniyet ile benimseme arasındaki açıklık, dil seçiminin salt teknik bir optimizasyon problemi olmadığını; öğrenme eğrisi, mevcut kod tabanlarının ağırlığı ve ekip yetkinliğinin bağımsız birer faktör olduğunu göstermektedir.

4.7. Uygulama Alanına Göre Dil Dağılımı

Sistem ve gömülü yazılım alanında C %60,8 ile birinci, C++ %55,5 ile üçüncü sıradadır; Assembly %21,9 ile yalnızca bu alanda anlamlı bir paya ulaşmaktadır. Bu alanda Python'un %60,2 ile ikinci sırada yer alması, gömülü geliştirmede de betikleme ve test otomasyonunun yaygınlığını yansıtmaktadır. Bu dağılım, donanıma yakın çalışmanın manuel bellek denetimi ve öngörülebilir çalışma zamanı gerektirdiği yönündeki tezi doğrulamaktadır. Dikkat çekici olan, Rust'ın bu alanda %18,1 ile diğer tüm alanlardaki oranının üzerine çıkmasıdır; Rust'ın benimsenmesi, tam da bellek güvenliği açıklarının en maliyetli olduğu alanda yoğunlaşmaktadır. Bu bulgu, Bölüm 1'de aktarılan Android deneyimiyle aynı yöne işaret etmektedir. Uygulama alanlarına göre dil dağılımı Şekil 7'de gösterilmekte, koyu renkler daha yüksek kullanım oranlarını temsil etmektedir.



Şekil 7. Uygulama alanına göre dil kullanım oranları.

Bulut ve altyapı alanında Go %28,1 ile zirve yapmaktadır; bu, Go'nun tüm alanlar içindeki en yüksek oranıdır (genel ortalaması %10,6). Bölüm 4.4'teki başlatma süresi ölçümü bu tercihin teknik zeminini oluşturmaktadır. Siber güvenlik alanında ise Python %62,7 ile birinci, Bash %52,2 ile ikinci

sıradadır; C ise %33,8 ile genel ortalamasının (%14,2) iki katından fazla bir orana ulaşmaktadır. Bu ikili dağılım alanın kendi iç bölünmesini yansıtmaktadır: betikleme ve otomasyon tarafında Python ve Bash, düşük seviye analiz ve açık araştırması tarafında ise C ve Assembly (%19,2) hâkimdir. Bu farklılaşma, aynı uygulama alanı içinde bile dil tercihinin yapılan işin niteliğine göre ayrıştığını göstermektedir. Uygulama alanlarına göre en yaygın diller Tablo 3'te özetlenmiş olup, yüzdelere her bir alan içindeki kullanım oranlarını göstermektedir.

Tablo 3. Uygulama alanına göre en yaygın beş programlama dili.

Uygulama alanı	n	1.	2.	3.	4.	5.
Yapay zekâ / ML	1.784	Python 62	SQL 40	Bash 35	JS 31	C++ 19
Sunucu / Backend	6.453	SQL 47	JS 41	Python 40	Bash 37	TS 27
Sistem / Gömülü	1.274	C 61	Python 60	C++ 56	Bash 51	Asm 22
Bulut / Altyapı	1.974	Bash 56	Python 54	SQL 43	JS 40	Go 28
Siber güvenlik	370	Python 63	Bash 52	JS 46	SQL 41	PS 35
Web / Frontend	14.325	JS 65	SQL 49	TS 49	Python 34	Bash 32
Kurumsal / ERP	1.919	SQL 44	JS 39	Python 38	C# 36	C++ 34
Veri mühendisliği	945	Python 62	SQL 61	Bash 41	JS 31	PS 19
Oyun / Grafik	451	C++ 48	C# 41	Python 36	JS 34	C 29
Mobil	1.391	Kotlin 38	Swift 35	JS 30	Java 28	Python 23

4.8. Çok Eksenli Karşılaştırma Matrisi

Tablo 4, incelenen dilleri altı eksenle özetlemekte ve dil seçiminin tek bir performans ölçütüne indirgenmeden değerlendirilmesine olanak sağlamaktadır. Performansa ilişkin üç sütun (hesaplama, tahsis, başlatma) doğrudan Bölüm 4.2–4.4'te raporlanan ölçüm sonuçlarından alınmıştır. Bellek güvenliği sütunu, dilin bellek güvenliğini sağlama mekanizmasını kategorik olarak belirtir. Ekosistem ve öğrenme eğrisi sütunları ise doğaları gereği niteldir ve aşağıdaki ölçütlere göre atanmıştır.

Ekosistem düzeyi, dilin paket yöneticisindeki kütüphane sayısı ve kurumsal kullanım yaygınlığı temel alınarak üç kademedeki değerlendirilmiştir: "Çok yüksek" (on yıldan uzun süredir yaygın kurumsal kullanım ve olgun standart kütüphane), "Yüksek" (büyüyen fakat henüz daha genç ekosistem), "Orta" (niş veya akademik ağırlıklı kullanım). Öğrenme eğrisi ise dilin temel üretkenliğe ulaşmak için gerektirdiği kavramsal yük temelinde "Kolay", "Orta" ve "Dik" olarak sınıflandırılmıştır; örneğin Rust'ın sahiplik ve ödünç alma modeli ile Haskell'in tembel değerlendirme ve tip sınıfı kavramları "Dik" kategorisini oluşturur. Rust'ın öğrenme zorluğunun ampirik olarak da belgelendiğini belirtmek gerekir; Zhu vd. (2022), yüz bir Rust geliştiricisiyle yaptıkları karma

yöntemli çalışmada sahiplik ve yaşam süresi (lifetime) kurallarının en sık zorlanılan konular olduğunu raporlamıştır. Bu nitel atamalar kesin ölçümler değil, yönlendirici değerlendirmelerdir.

Tablo 4. Ağ iletişimi ve güvenlik odaklı çok eksenli dil karşılaştırması.

Dil	Bellek güvenliği	Hesaplama (ms)	Tahsis (ms)	Başlatma (ms)	Ekosistem	Öğrenme
Rust	Derleme zamanı	97,8	21,6	1,70	Yüksek	Dik
Go	Çöp toplama (izleyici)	80,9	38,0	1,83	Yüksek	Kolay
Haskell	Çöp toplama (izleyici)	1.703,2	13,7	2,18	Orta	Dik
C	Yok (manuel)	58,6	8,8	1,49	Çok yüksek	Orta
C++	Manuel (RAII ile kısmi)	58,5	8,9	1,48	Çok yüksek	Dik
Java	Çöp toplama (izleyici)	173,6	31,7	24,43	Çok yüksek	Orta
Python	Referans sayma	30.191,3	306,0	16,46	Çok yüksek	Çok kolay

4.9. Vaka İncelemesi: Rustls ve OpenSSL

"Bellek güvenli diller yavaştır" önyargısı, uzun süre C/C++'tan geçişin önündeki başlıca gerekçelerden biri olmuştur. Ancak son yıllardaki ölçümler bu varsayımı önemli ölçüde zayıflatmıştır. ISRG tarafından desteklenen ve Rust ile yazılmış Rustls TLS kütüphanesi, sunucu tarafı el sıkışma gecikmesinde OpenSSL'in yaklaşık yarısı gecikmeyle çalışmakta ve seksen çekirdeğe kadar neredeyse doğrusal ölçeklenmektedir; yeniden kullanılan el sıkışmalarda %80 ilâ %330 arasında daha yüksek verim sağladığı raporlanmıştır (Prossimo, 2025). Bu sonuç, en azından TLS gibi olgun kütüphanelerin bulunduğu alanlarda güvenlik ile performans arasında bir seçim yapmanın zorunlu olmadığını göstermektedir. Bu bulgu, Bölüm 4.3'teki tahsis ölçümüyle aynı yöne işaret etmektedir.

5. Türkiye İçin Dil Seçim ve Geçiş Çerçevesi

Elde edilen bulgular, Türkiye'deki kritik altyapı projelerinde bileşen bazlı bir dil seçim çerçevesi için zemin oluşturmaktadır. Güvenlik açısından kritik çekirdek bileşenler (protokol yığınları, ayrıştırıcılar, kriptografik katmanlar) için Rust; ölçeklenebilir servisler, API ağ geçitleri ve mikroservisler için Go; otomasyon, test ve veri işleme için Python; mevcut kurumsal sistemlerle entegrasyon gerektiren katmanlar için ise ekosistem olgunluğu nedeniyle Java önerilmektedir. Bu

öneriler ölçülen başarımlar ve bellek güvenliği özelliklerine dayanmakta olup, biçimsel doğrulama gerektiren niş bileşenler bu çalışmanın kapsamı dışındadır.

Bu öneriler, Bölüm 4.2–4.5'teki ölçümlerin yanı sıra Bölüm 4.7'deki anket bulgularına da dayanmaktadır. Örneğin bulut ve altyapı alanında Go'nun %28,1 ile zirve yapması, bu alandaki fiili tercihlerin başlatma süresi ölçümüyle (Bölüm 4.4) tutarlı olduğunu göstermektedir. Benzer biçimde, Rust'ın sistem ve gömülü alanda kendi zirvesine (%18,1) ulaşması, güvenlik kritik çekirdek için Rust önerisiyle tutarlıdır.

Stack Overflow 2025 veri setinde ülke bilgisini Türkiye olarak bildiren 295 yanıtlayıcı bulunmaktadır. Bu alt örneklem küresel örnekleme kıyasla küçük olduğundan aşağıdaki karşılaştırma kesin bir ölçüm değil, eğilim göstergesi olarak değerlendirilmelidir. Söz konusu alt örnekleme dil dağılımı küresel dağılımdan ayrılmaktadır. Türkiye'deki yanıtlayıcılar ortalama olarak daha fazla dil bildirmiş; incelenen on üç dilin toplam kullanım oranı Türkiye'de %356,6 iken küresel örnekleme %283,5'tir. Bu nedenle karşılaştırma ham yüzdeler yerine göreceli oran (Türkiye ÷ küresel) üzerinden yapılmıştır. Söz konusu ölçüte göre C ($\times 1,65$), C# ($\times 1,75$) ve Kotlin ($\times 1,89$) Türkiye'de belirgin biçimde daha yaygınken, Rust $\times 0,99$ ile küresel düzeyde kalan tek dildir (%9,5'e karşı %9,6). Türkiye ile küresel örneklem arasındaki dil kullanım farkları Tablo 5'te gösterilmektedir.

Tablo 5. Türkiye ve küresel örnekleme programlama dili kullanım oranlarının karşılaştırılması.

Dil	Türkiye (%)	Küresel (%)	Göreceli oran
Kotlin	13,2	7,0	1,89
C#	31,5	18,0	1,75
C	23,4	14,2	1,65
Go	15,6	10,6	1,47
C++	20,3	15,2	1,34
JavaScript	54,6	42,7	1,28
SQL	47,8	37,9	1,26
Java	23,7	19,0	1,25
Python	44,1	37,4	1,18
TypeScript	31,5	28,2	1,12
PHP	13,6	12,2	1,11
Rust	9,5	9,6	0,99
Bash/Shell	27,8	31,5	0,88

Bu dağılım, bellek güvenliği açısından çift yönlü bir tablo ortaya koymaktadır. Bir yandan manuel bellek yönetimine dayanan C'nin Türkiye'deki kullanım oranı küresel ortalamasının 1,65 katıdır; diğer yandan bellek güvenli bir alternatif olan Rust'ın benimsenmesi küresel düzeyin ötesine

geçmemiştir. Bu durum, Türkiye'de bellek güvenli dillere geçiş için hem ihtiyacın hem de potansiyelin bulunduğuna işaret etmektedir.

Geçiş, mevcut kod tabanının tümüyle yeniden yazılması biçiminde değil, kademeli olarak tasarlanmalıdır. Google'ın Android deneyimi, eski kodun büyük bölümü değiştirilmeden yalnızca yeni kodun bellek güvenli dillerle yazılmasının dahi açık oranını %76'dan %24'e indirebildiğini göstermektedir (Bölüm 1). Bu yaklaşım, sınırlı kaynakla çalışan kurumlar için gerçekçi bir yol haritası sunmaktadır: önce yeni geliştirmelerde bellek güvenli dile geçilmesi, ardından en yüksek risk taşıyan bileşenlerin önceliklendirilerek dönüştürülmesi.

Geçişin başarısı ekip yetkinliğinin geliştirilmesine, araç zinciri yatırımına ve birlikte çalışabilirlik (FFI) katmanlarının doğru tasarlanmasına bağlıdır. Olası engeller ise öğrenme eğrisi, nitelikli insan kaynağı kısıtı ve mevcut sistemlerle entegrasyon maliyetidir; Bölüm 4.6'da raporlanan yüksek sadakat oranı, öğrenme eğrisi aşıldığında geliştiricilerin dili kullanmaya devam etme eğiliminde olduğuna işaret etmektedir.

6. Sınırlılıklar

Bulgular aşağıdaki sınırlılıklar çerçevesinde değerlendirilmelidir.

Deney tek çekirdekli bir ortamda yürütülmüştür; bu nedenle Go'nun goroutine modeli ile Rust ve Java'nın paralel ölçekleme avantajı gözlenememiştir. Sunucu ve ağ iş yüklerine ilişkin değerlendirmeler bu yönüyle eksiktir ve çok çekirdekli ölçümle tamamlanmalıdır. Ayrıca mikro-ölçümler belirli maliyet bileşenlerini izole eder; uçtan uca sistem davranışını temsil etmez. Aynı algoritmanın diller arası eşdeğer uygulaması, Haskell gibi bazı dillerin deyimisel olmayan biçimde kullanılmasına yol açmakta ve bu dillerin en iyi hâlini yansıtmayabilmektedir. Bu nedenle elde edilen performans farkları, kullanılan donanım-yazılım yapılandırması ve seçilen mikro-iş yükleri bağlamında yorumlanmalı; farklı sistemlerde aynı büyüklükte farkların ortaya çıkacağı varsayılmamalıdır.

Ölçümler WSL2 üzerinde çalışan sanallaştırılmış bir ortamda yürütülmüştür; bu durum özellikle soğuk başlatma ve ağ gecikmesi ölçümlerinde fiziksel donanıma kıyasla ek bir değişkenlik kaynağı oluşturabilir. Raporlanan dayanıklı değişim katsayıları ölçüm içi kararlılığı desteklemekle birlikte, mutlak değerlerin farklı donanımlarda yinelenmesi hâlinde farklılaşması beklenmelidir.

Ölçüm uygulamalarında diller arası tam eşdeğerlik her boyutta sağlanamamıştır. Tahsisin derleyici tarafından elenmesini önleyen mekanizmalar dillere göre değişmektedir: Rust'ta `black_box`, Go'da `runtime.KeepAlive` kullanılmış, C ve C++'ta `free` çağrısı bu işlevi doğal olarak görmüş, yönetilen dillerde ise ek bir bariyer kullanılmamıştır. Benzer biçimde tahsis iş yükünde yönetilen dillerin çalışma zamanları ayrılan belleği sıfırlarken, manuel yönetim uygulayan C ve C++ bu işlemi

yapmamaktadır. Söz konusu farklar ölçülen bellek yönetimi maliyetinin doğal bileşenleri olarak değerlendirilmiş, ancak sonuçların yorumlanmasında göz önünde bulundurulmalıdır. Ağ ölçümünde ise TCP_NODELAY seçeneği tüm dillerde açıkça ayarlanmıştır.

PHP'nin hesaplama ölçümünde yirmi turun üçü, geri kalanların yaklaşık 1,8 katı bir süre vermiştir. Söz konusu turlar birbirine çok yakın değerler ürettiğinden bu dağılım rastgele gürültüden ziyade ortam kaynaklı dönemsel bir etkiye işaret etmektedir. Aykırı turlar hesaplama çıkartılmamış, medyan kullanımı sayesinde merkezi eğilim tahmininin bu turlardan etkilenmemesi sağlanmıştır.

Veri seti gönüllü katılıma dayanmakta ve öz-seçim yanlılığı taşımaktadır; Stack Overflow örnekleme platformun kullanıcı tabanına eğilimlidir. Uygulama alanları bildirilen birincil iş rolü üzerinden tanımlandığından, birden çok alanda çalışan geliştiriciler tek alana atanmıştır. Ayrıca "kullandım" ifadesi kullanım yoğunluğunu değil varlığını ölçmektedir. Son olarak Stack Overflow 2025 anketinin dil listesinde Haskell yer almadığından, Bölüm 4.2'de ölçülen Haskell performansı anket verisiyle eşleştirilememiştir. Ayrıca Türkiye alt örnekleme küresel örnekleme kıyasla küçüktür ve Stack Overflow kullanıcı tabanının Türkiye'deki geliştirici nüfusunu ne ölçüde temsil ettiği bilinmemektedir; bu nedenle Bölüm 5'teki karşılaştırma kesin bir ölçüm değil, eğilim göstergesi olarak değerlendirilmelidir.

Dil seçimi güvenliğin yalnızca bir bileşenidir. Bellek güvenli bir dil; mantık hatalarını, yanlış yapılandırmaları, zayıf kriptografik seçimleri ya da tedarik zinciri risklerini ortadan kaldırmaz. Bellek güvenli diller tek başına yeterli bir çözüm oluşturmaz; hatalı kodlama pratikleri her dilde açık üretebilir. Dolayısıyla dil seçimi, kapsamlı bir güvenli geliştirme yaşam döngüsünün yerine değil, onun içinde değerlendirilmelidir. Son olarak dil ekosistemleri hızla evrildiğinden, bugün geçerli olan bir değerlendirme birkaç yıl içinde değişebilir.

7. Sonuçlar ve Öneriler

Çalışmanın başında ortaya konan dört araştırma sorusu, elde edilen bulgular ışığında aşağıdaki gibi yanıtlanmaktadır.

AS1'e ilişkin olarak, bellek güvenliği garantisinin zorunlu bir performans maliyeti getirmediği görülmüştür. Rust'ın sahiplik modeli, tahsis yoğun iş yükünde Go'nun çöp toplayıcısından 1,76 ve Java'nınkinden 1,47 kat hızlı çalışmıştır. Buna karşılık yine çöp toplayıcı kullanan Haskell (13,7 ms) ve C# (18,7 ms) Rust'ın (21,6 ms) önüne geçmiştir; bu da belirleyici olanın denetimin derleme zamanında mı çalışma zamanında mı yapıldığı değil, tahsis mekanizmasının kendisi olduğunu göstermektedir. Hesaplama ekseninde ise Rust, C'nin yaklaşık 1,67 katı süreyle çalışmıştır; bu fark

büyük ölçüde dizi sınır denetimlerinden kaynaklanmaktadır. Dolayısıyla güvenliğin bedeli sıfır değil, ancak ölçülebilir ve sınırlıdır.

AS2'ye ilişkin olarak, yürütme modelinin soğuk başlatma üzerindeki etkisi belirleyicidir. Makine koduna derlenen diller 2,2 ms'nin altında başlatılırken JVM 24,4 ms gerektirmiştir; aradaki yaklaşık on altı katlık fark, uzun ömürlü sunucu süreçlerinde ihmal edilebilirken serverless fonksiyonlar, komut satırı araçları ve kenar hizmetlerinde doğrudan belirleyici hâle gelmektedir.

AS3'e ilişkin olarak, geliştiricilerin fiili tercihleri ölçüm bulgularıyla büyük ölçüde tutarlıdır. Go'nun bulut ve altyapı alanında %28,1 ile kendi zirvesine ulaşması başlatma süresi avantajıyla; C'nin sistem ve gömülü alanda %60,8 ile birinci olması ham hesaplama hızıyla; Rust'ın aynı alanda kendi en yüksek oranına (%18,1) çıkması ise bellek güvenliği açıklarının bu alandaki yüksek maliyetiyle örtüşmektedir. Bununla birlikte tam bir örtüşme yoktur: sunucu ve backend alanında teknik olarak öne çıkan diller yerine SQL, JavaScript ve Python'un baskın olması, mevcut kod tabanlarının ve ekosistem bağımlılıklarının teknik uygunluktan bağımsız bir belirleyici olduğunu göstermektedir.

AS4'e ilişkin olarak, teknik üstünlük ve geliştirici memnuniyeti benimsemeyi açıklamakta tek başına yeterli değildir. Rust %72,5 ile en yüksek sadakat oranına ve +4.538 ile en yüksek net talebe sahip olmasına karşın fiili kullanım oranı %9,6'da kalmıştır. Bu açıklık, öğrenme eğrisi, mevcut kod tabanlarının ağırlığı ve ekip yetkinliğinin bağımsız birer engel oluşturduğunu göstermekte; dil seçiminin salt teknik bir optimizasyon problemi olmadığını ortaya koymaktadır.

İncelenen diller arasında evrensel bir "en iyi" bulunmamaktadır; doğru seçim, bileşenin güvenlik hassasiyetine, performans gereksinimine ve ekibin yetkinliğine göre değişmektedir. Rust, güvenlik ile performansı en dengeli biçimde buluşturarak kritik çekirdek için öne çıkmaktadır; Go pratiklik ve ölçeklenebilirlik sunmakta, Haskell ise bu ölçümlerde en yavaş gruplardan birinde yer almakla birlikte farklı bir tasarım yaklaşımını temsil etmektedir. C/C++ performansta zirveyi korurken güvenlik maliyeti yüksektir; Python ve Java ise kendi alanlarında üretkenlik ve olgunluk sağlamaktadır.

Çalışmanın ampirik katkısı üç noktada toplanmaktadır. Birincisi, on bir dilde yürütülen kontrollü ölçüm, Rust'ın sahiplik modelinin tahsis yoğun iş yükünde Go'nun çöp toplayıcısından 1,76 kat hızlı olduğunu göstermiş; aynı ölçüm, kuşak tabanlı çöp toplayıcı kullanan Haskell ve C#'ın Rust'ın önüne geçmesiyle, belirleyici olanın bellek yönetimi kategorisi değil tahsis mekanizması olduğunu ortaya koymuştur. İkincisi, soğuk başlatma ölçümü makine koduna derlenen diller ile JVM arasında yaklaşık on altı katlık bir fark ortaya koymuş; ağ ölçümü ise Rust'ın gecikme ve verim bakımından C ile aynı düzeyde yer aldığını, çöp toplamalı dilleri ise geride bıraktığını göstererek kenar bilişim ve ağ hizmetlerinde dil seçiminin teknik zeminini netleştirmiştir. Üçüncüsü, 49.191 yanıtlayıcılık anketin ham verisi üzerinde yapılan analiz, Rust'ın %72,5 ile en yüksek geliştirici

sadakatine sahip olduğunu, buna karşılık fiili kullanımının %9,6'da kaldığını göstermiş; teknik üstünlüğün tek başına benimsemeyi getirmediğini ortaya koymuştur.

Türkiye alt örnekleminde ortaya çıkan tablo bu sonucu desteklemektedir: manuel bellek yönetimine dayanan C'nin kullanım oranı küresel ortalamanın 1,65 katı iken, Rust'ın benimsenmesi küresel düzeyin ötesine geçmemiştir.

Türkiye'nin siber dayanıklılığını güçlendirmek için bellek güvenli dillere yapılacak stratejik yatırım, yalnızca bir teknoloji tercihi değil, ulusal güvenlik tedbiri niteliğindedir. Karma mimari ve kademeli geçiş yaklaşımı, bu dönüşümü gerçekçi ve risk yönetimli biçimde hayata geçirmenin en uygulanabilir yoludur.

8. Gelecek Çalışmalar

En öncelikli uzantı, ölçüm deneyinin çok çekirdekli bir ortamda yinelenerek paralel ölçekleme davranışının sayısallaştırılmasıdır. Mevcut çalışmada tüm ölçümler tek çekirdeğe sabitlendiğinden, Go'nun goroutine modeli ile Rust'ın iş parçacığı güvenliği garantilerinin ölçekleme altındaki davranışı gözlenememiştir; bu, ağ ve sunucu bağlamında doğrudan karşılığı olan bir boşluktur.

İkinci bir eksen, tahsis bulgularının derinleştirilmesidir. Bu çalışmada kuşak tabanlı çöp toplayıcı kullanan Haskell ve C#, derleme zamanı denetimine dayanan Rust'ın önüne geçmiştir. Söz konusu farkın hangi ölçüde fidelik tahsisinden, hangi ölçüde nesne yaşam süresi dağılımından kaynaklandığı, farklı yaşam süresi profillerine sahip iş yükleriyle ayrıştırılabilir. Benzer biçimde, Java'nın tahsis ölçümünde gözlenen yüksek değişkenliğin (rCV %10,9) duraklama süresi dağılımı olarak doğrudan ölçülmesi, gecikme güvencesi gerektiren sistemler için işlevsel bir katkı sağlayacaktır.

Üçüncü olarak, ağ ölçümünün geri döngü arayüzü yerine gerçek bir ağ üzerinde yinelenmesi önerilmektedir. Mevcut bulgularda yorumlanan dillerin gecikme ekseninde çöp toplamalı dilleri geçmesi, işin büyük ölçüde çekirdeğe devredilmesiyle açıklanmıştır; fiziksel gecikmenin eklendiği koşullarda bu farkın ne ölçüde korunduğu sınanmalıdır. Ayrıca ölçümün eşzamanlı çok sayıda bağlantıyla tekrarlanması, tek bağlantılı yankı deseninin gizlediği zamanlayıcı maliyetlerini açığa çıkarabilir.

Dördüncü bir eksen, Türkiye'de manuel bellek yönetimine dayanan dillerin görece yaygın olmasının nedenleridir. Eğitim müfredatı, mevcut kod tabanlarının yapısı ve sektörel dağılım gibi etkenler nitel yöntemlerle incelenmeye değer bir araştırma konusu oluşturmaktadır. Buna ek olarak, Türkiye'deki kritik altyapı projelerinden gerçek vaka çalışmalarının derlenmesi ve geçişin maliyet-fayda analizinin yerel ekonomik verilerle modellenmesi önerilmektedir.

Son olarak, biçimsel doğrulama araçlarının (Rust için Prusti, Haskell için LiquidHaskell) güvenlik kritik bileşenlerde uygulanabilirliği ile üretken yapay zekâ araçlarının bellek güvenli dillerde ne ölçüde doğru ve güvenli kod ürettiği, geçiş stratejilerini doğrudan etkileyecek araştırma alanları oluşturmaktadır.

Ek A. Ölçüm Deneyinin Yeniden Üretilmesi

Bölüm 4.2–4.4'te raporlanan ölçümlerin bağımsız olarak yinelenebilmesi için gerekli tanımlar, sürüm bilgileri ve komutlar aşağıda verilmektedir.

A.1. İş Yüklerinin Tanımı

Birinci iş yükü (`buffer_scan`) 16 MB'lık bir tampon üzerinde on tur boyunca her bayta bir bit sola döndürme uygular ve sonucu bayt indisıyla XOR'layarak biriktirir. Tampon, i indisli her bayt için $(i \times 31 + 7) \bmod 256$ formülüyle belirlenimci biçimde doldurulur; böylece rastgelelik kaynağı bulunmaz ve her çalıştırma aynı sonucu verir. İkinci iş yükü (`alloc_churn`) 16 ilâ 255 bayt arasında değişen boyutlarda bir milyon tahsis gerçekleştirir, her bloğun ilk ve son baytına yazar ve bu değerleri biriktirir. Tahsis boyutunun değişken tutulması, sabit boyutlu havuz optimizasyonlarının ölçümü çarpıtmasını engellemek içindir.

Her iki iş yükü de bir sağlama değeri üretmektedir. Doğruluk denetimi olarak on bir uygulamanın tamamı `buffer_scan` için 21.390.950.400, `alloc_churn` için 253.384.800 değerini üretmiştir. Bu denetim her ölçüm turunda yinelenmiş ve ekteki ham kayıtlara işlenmiştir. Böylece yalnızca çalışma sürelerinin değil, farklı dillerde gerçekleştirilen işlemlerin işlevsel olarak eşdeğerliğinin de denetlenmesi amaçlanmıştır. Söz konusu denetim, dillerden hiçbirinin döngüyü kısaltarak, işlemi atlayarak veya derleyici tarafından tümüyle optimize edilerek haksız avantaj kazanmadığını göstermektedir. Sağlama değeri tutmayan bir uygulama karşılaştırmaya dâhil edilmemiş ve bu nedenle raporlanan ölçümler yalnızca doğruluk denetimini geçen uygulamalardan elde edilmiştir.

A.2. Derleme ve Çalıştırma

Ölçüm paketinin tamamına <https://github.com/muhammedSeyrek/bellek-guvenligi-olcum> adresinden erişilebilmektedir. Kullanılan çalışma zamanı sürümleri ve derleme komutları Tablo A.1'de verilmiştir. Bu bilgiler, ölçüm ortamının aynı yazılım sürümleriyle yeniden kurulabilmesi ve deneylerin bağımsız olarak yinelenebilmesi amacıyla sunulmuştur.

Tablo A.1. Ölçümde kullanılan çalışma zamanı sürümleri ve derleme komutları.

Dil	Sürüm	Derleme / çalıştırma komutu
C	GCC 13.3.0	gcc -O2 -o bin_c bench.c
C++	GCC 13.3.0	g++ -O2 -o bin_cpp bench.cpp
Rust	1.75.0	rustc -O -o bin_rs bench.rs
C#	.NET 8.0.131	dotnet build -c Release
Go	1.22.2	go build -o bin_go bench.go
Java	OpenJDK 21.0.12	javac Bench.java && java Bench
Haskell	GHC 9.4.7	ghc -O2 -o bin_hs bench.hs
JavaScript	Node.js 22.23.3	node bench.js
PHP	8.3.6	php bench.php
Ruby	3.2.3	ruby bench.rb
Python	3.12.3	python3 bench.py

Ölçümler, tüm süreçlerin taskset ile tek bir çekirdeğe sabitlendiği bir Ubuntu 24.04 ortamında alınmıştır. Her ölçüm yirmi kez tekrarlanmış, ilk tur ısınma olarak atılmış ve medyan değer raporlanmıştır. Soğuk başlatma süresi, ilgili dilde yalnızca çıkış yapan boş bir programın aynı biçimde yirmi kez çalıştırılmasıyla ölçülmüştür; ölçüm sarmalayıcısının sabit ek yükü /bin/true çalıştırılarak ayrıca belirlenmiş ve 1,48 ms olarak kaydedilmiştir. Ölçüm ve istatistik hesaplama işlemlerinin tamamı, çalışmanın ekinde sunulan olcum_calistir.py, baslatma_calistir.py ve ag_calistir.py betikleriyle otomatikleştirilmiştir.

Ek B. Anket Verisinin Analizi

B.1. Veri Kaynakları

Bölüm 4.6, 4.7 ve 5'teki bulgular, kamuya açık bir veri setinin ham mikro-verisine dayanmaktadır. Stack Overflow Annual Developer Survey 2025 veri seti Open Database License (ODbL) kapsamında yayımlanmakta olup <https://github.com/StackExchange/Survey/tree/main/packages/archive/2025> adresinden indirilebilmektedir (Erişim tarihi: 8 Ekim 2026); analizde results.csv dosyası kullanılmış ve 49.191 geçerli yanıt işlenmiştir. Bölüm 5'teki Türkiye karşılaştırması, aynı veri setinde ülke bilgisi Türkiye olarak bildirilen $n = 295$ yanıtlayıcıdan oluşan alt örneklem üzerinde hesaplanmıştır. Veri seti yanıtlayıcı düzeyinde anonimleştirilmiştir ve kişisel veri içermemektedir; bu nedenle çalışma için etik kurul onayı gerekmemektedir.

B.2. Göstergelerin Tanımı

Diller ankette çoklu-seçim biçiminde sorulduğundan, bir yanıtlayıcı birden çok dil işaretleyebilmektedir; bu nedenle yüzdeler örneklem büyüklüğüne oranlanmakta ve toplamaları %100'ü aşmaktadır. Analizde kullanılan üç gösterge şu şekilde tanımlanmaktadır.

Kullanım oranı, ilgili dili son bir yıl içinde kullandığını bildirenlerin tüm örnekleme oranıdır. Sadakat oranı, dili kullananlar içinde o dile hayranlık duyduğunu bildirenlerin payıdır; paydası tüm örneklem değil yalnızca o dilin kullanıcılarıdır ve dilin kullanıcılarını elde tutma gücünü ölçer. Net talep, dili gelecek yıl kullanmak isteyenlerin sayısı ile hâlihazırda kullananların sayısı arasındaki farktır; pozitif değer dilin mevcut kullanıcı tabanından daha geniş bir ilgi gördüğünü, negatif değer ise dilin büyük ölçüde zorunluluktan kullanıldığını göstermektedir. Bu üç göstergenin paydaları farklı olduğundan doğrudan birbirine oranlanmamalıdır.

Uygulama alanları, yanıtlayıcının bildirdiği birincil iş rolü (DevType alanı) üzerinden tanımlanmıştır. Örneğin bulut ve altyapı alanı; bulut altyapı mühendisleri, DevOps uzmanları ve sistem yöneticilerini kapsamaktadır.

B.3. Analiz Betiği

Yukarıda tanımlanan tüm göstergeler ve çapraz tablolar, çalışmanın ekinde sunulan anket_analizi.py betiği ile üretilmektedir. Betik, veri setlerinin yerel kopyalarını girdi olarak alır ve makalede raporlanan tabloların tamamını CSV biçiminde yeniden üretir. Betiğin çalıştırılması için pandas ve numpy kütüphaneleri yeterlidir; veri setleri lisansları gereği betikle birlikte dağıtılmamakta, yukarıda verilen adreslerden indirilmesi gerekmektedir.

Ek C. Sunulan Dosyalar

Ölçüm ve analiz süreçlerinde kullanılan tüm kaynak dosyalar ile ara çıktılar Tablo C.1'de listelenmekte olup, tamamı kalıcı erişim için Zenodo üzerinde yayımlanmıştır (Yıldırım ve Seyrek, 2026).

Tablo C.1. Çalışmanın eki olarak sunulan dosyalar.

Dosya	İçerik	İlgili bölüm
README.txt	Ek paketin açıklaması ve içerik listesi	—
OKU.md	Ölçümün yeniden yürütülmesi için yönerge	3.1, Ek A
olcum_calistir.py	Hesaplama ve tahsis ölçüm betiği	4.2–4.3

Dosya	İçerik	İlgili bölüm
baslatma_calistir.py	Soğuk başlatma ve ikili boyut ölçüm betiği	4.4
ag_calistir.py	TCP yankı ağ ölçüm betiği	4.5
anket_analizi.py	Anket analiz betiği	4.6–4.7, 5
sekilller.py	Şekil üretim betiği	4.2–4.5
kod_hesaplama/bench.*	On bir dilde buffer_scan ve alloc_churn kodu	4.2–4.3
kod_baslatma/bos.*	On bir dilde boş program	4.4
kod_ag/net_*	Sekiz dilde TCP yankı sunucu ve istemci kodu	4.5

Yazarların Katkısı

Tüm yazarlar çalışmaya eşit katkıda bulunmuştur.

Çıkar Çatışması Beyanı

Yazarlar arasında herhangi bir çıkar çatışması bulunmamaktadır.

Araştırma ve Yayın Etiği Beyanı

Yapılan çalışmada araştırma ve yayın etiğine uyulmuştur.

Kaynaklar

- Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A. J. (2020). How do programmers use unsafe Rust? Proceedings of the ACM on Programming Languages, 4(OOPSLA), 1-27. <https://doi.org/10.1145/3428204>
- Bugden, W., Alahmar, A. (2022). The safety and performance of prominent programming languages. International Journal of Software Engineering and Knowledge Engineering, 32(5), 713-744. <https://doi.org/10.1142/S0218194022500231>
- Consortium for Information & Software Quality (CISQ). (2022). The cost of poor software quality in the US: A 2022 report. Needham, MA: CISQ.
- Cybersecurity and Infrastructure Security Agency (CISA), National Security Agency (NSA), and Federal Bureau of Investigation (FBI). (2023). The case for memory safe roadmaps. Washington, DC: CISA.
- Defense Advanced Research Projects Agency (DARPA). (2024). TRACTOR: Translating all C to Rust. Arlington, VA: DARPA.
- Evans, A. N., Campbell, B., Soffa, M. L. (2020). Is Rust used safely by software developers? Proceedings of the 42nd International Conference on Software Engineering (ICSE '20), 246-257. <https://doi.org/10.1145/3377811.3380413>
- Li, H., Guo, L., Yang, Y., Wang, S., Xu, M. (2024). An empirical study of Rust-for-Linux: The success, dissatisfaction, and compromise. Proceedings of the 2024 USENIX Annual Technical Conference (ATC '24), 425-443. Erişim adresi: <https://www.usenix.org/conference/atc24/presentation/li-hongyu>
- MITRE Corporation. (2024). 2024 CWE top 25 most dangerous software weaknesses. McLean, VA: MITRE.

- Nanz, S., Furia, C. A. (2015). A comparative study of programming languages in Rosetta Code. Proceedings of the 37th International Conference on Software Engineering (ICSE '15), 778-788. <https://doi.org/10.1109/ICSE.2015.90>
- National Security Agency (NSA). (2025). Memory safe languages: Reducing vulnerabilities in modern software development (Cybersecurity Information Sheet). Fort Meade, MD: NSA.
- Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., Saraiva, J. (2017). Energy efficiency across programming languages: How do energy, time, and memory relate? Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE '17), 256-267. <https://doi.org/10.1145/3136014.3136031>
- Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., Saraiva, J. (2021). Ranking programming languages by energy efficiency. Science of Computer Programming, 205, 102609. <https://doi.org/10.1016/j.scico.2021.102609>
- Prechelt, L. (2000). An empirical comparison of seven programming languages. IEEE Computer, 33(10), 23-29. <https://doi.org/10.1109/2.876288>
- Prossimo. (2025). Rustls performance report: Comparison with OpenSSL and BoringSSL. San Francisco, CA: Internet Security Research Group.
- Qin, B., Chen, Y., Yu, Z., Song, L., Zhang, Y. (2020). Understanding memory and thread safety practices and issues in real-world Rust programs. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20), 763-779. <https://doi.org/10.1145/3385412.3386036>
- Stack Overflow. (2025). Stack Overflow annual developer survey 2025 [Veri seti]. Open Database License. Erişim adresi: <https://github.com/StackExchange/Survey/tree/main/packages/archive/2025> (Erişim tarihi: 8 Ekim 2026)
- The White House. (2024). Back to the building blocks: A path toward secure and measurable software. Washington, DC: Office of the National Cyber Director.
- Vander Stoep, J., Rebert, A. (2024). Eliminating memory safety vulnerabilities at the source. Google Security Blog. Erişim adresi: <https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html> (Erişim tarihi: 8 Ekim 2026)
- Xu, H., Chen, Z., Sun, M., Zhou, Y., Lyu, M. R. (2021). Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. ACM Transactions on Software Engineering and Methodology, 31(1), 1-25. <https://doi.org/10.1145/3466642>
- Yıldırım, M., Seyrek, M. (2026). Programlama dillerinde bellek güvenliği ve ağ iletişimi: Ölçüm paketi (Sürüm 1.0.0) [Yazılım ve veri seti]. Zenodo. <https://doi.org/10.5281/zenodo.23243615>
- Zhang, Y., Zhang, Y., Portokalidis, G., Xu, J. (2022). Towards understanding the runtime performance of Rust. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), Article 140. <https://doi.org/10.1145/3551349.3559544>
- Zheng, X., Wan, Z., Zhang, Y., Chang, R., Lo, D. (2024). A closer look at the security risks in the Rust ecosystem. ACM Transactions on Software Engineering and Methodology, 33(2), 1-30. <https://doi.org/10.1145/3624738>
- Zhu, S., Zhang, Z., Qin, B., Xiong, A., Song, L. (2022). Learning and programming challenges of Rust: A mixed-methods study. Proceedings of the 44th International Conference on Software Engineering (ICSE '22), 1269-1281. <https://doi.org/10.1145/3510003.3510164>