# LLM-QA : From Warnings to Safety Nets


Muhammet Ali BULUT
*Technology Faculty*
*Software Engineering*
*Firat University*
Elazığ, Türkiye
210541063@firat.edu.tr

Mine KILIÇ
*Technology Faculty*
*Software Engineering*
*Firat University*
Elazığ, Türkiye
240541136@firat.edu.tr

Esra GÜMÜŞ
*Technology Faculty*
*Software Engineering*
*Firat University*
Elazığ, Türkiye
210541063@firat.edu.tr

Kübra SOYSAL
*Technology Faculty*
*Software Engineering*
*Firat University*
Elazığ, Türkiye
240541158@firat.edu.tr

Elif ATAŞ
*Technology Faculty*
*Firat University*
*Software Engineering*
Elazığ, Türkiye
240541156@firat.edu.tr

Elif Eslem ÖZKAN
*Technology Faculty*
*Firat University*
*Software Engineering*
Elazığ, Türkiye
220541002@firat.edu.tr

Muhammet BAYKARA
*Technology Faculty*
*Firat University*
*Software Engineering*
Elazığ, Türkiye
mbaykara@firat.edu.tr



***Özet*— Modern Sürekli Entegrasyon/Sürekli Dağıtım (CI/CD) boru hatları, yazılım kalitesini sürdürmek için büyük ölçüde Statik Kod Analizi (SCA) araçlarına güvenmektedir. Ancak geleneksel SCA araçları, genellikle yüksek hatalı pozitif oranlarından muzdariptir ve "eyleme geçirilebilirlik açığı" (actionability gap) yaratan geçici uyarılar üretir. Bu makale, herhangi bir statik analiz aracına bağımlı kalmaksızın, ham kaynak kodunu doğrudan analiz ederek karmaşık mantık kusurlarını somut ve yürütülebilir birim testlerine dönüştüren, Büyük Dil Modelleri (LLM) tabanlı yeni bir "Kusurdan-Teste" (Defect-to-Test - D2T) çerçevesi olan LLM-QA'yı sunmaktadır. LLM-QA, devasa bir kod tabanını tek bir kanıtlanabilir test dosyasına daraltan, ölçeklenebilir ve durum bilgisiz (stateless) "The Funnel" (Huni) stratejisini kullanır. Sistem; Yapı Haritalama, Gözcü, Denetçi ve Hata Doğrulayıcı olmak üzere birbirini besleyen dört aşamalı bir istem (prompt) zinciri üzerinden çalışmaktadır. Bu süreçte Büyük Dil Modelleri, geleneksel araçların aksine "bağlamsal bir denetçi" rolü üstlenerek hata modellerini derinlemesine bir anlamsal süzgeçten geçirir ve Yapısal Düşünce Zinciri (SCoT) yöntemiyle her bir kusur için o hatayı somut olarak yeniden üretebilen bir "Kanıtlayıcı Test" (Proving Test) sentezler. Önerilen bu metodoloji, yalnızca kusurların tespit edilmesini sağlamakla kalmaz; aynı zamanda sentezlenen testleri otomatik olarak CI/CD boru hattına ve Git dallarına entegre ederek yazılım projeleri için sürdürülebilir bir varlık oluşturur. Böylelikle yazılım kalite güvencesi, hataları sadece bildiren pasif bir yapıdan, onları kalıcı bir "güvenlik ağı" (safety net) içinde kontrol altına alan aktif ve kanıta dayalı bir savunma mekanizmasına evrilir.**



***Keywords*— *Code Quality, Large Language Model, LLM, Software Quality Assurance, QA, Automated Test Generation, Static Code Analysis, SCA, Continuous Integration, Continuous Development, CI/CD***


Bu çalışmanın devamında

## I. GIRIŞ

Yazılım geliştirme dünyası, çevik metodolojiler ve Sürekli Entegrasyon/Sürekli Dağıtım (CI/CD) süreçlerinin etkisiyle hızlı yineleme döngülerine dayalı bir paradigmaya evrilmiştir. Bu hız baskısı altında yazılım kalitesini korumak için Statik Kod Analizi (SCA), programı çalıştırmadan kusurları tespit eden temel bir savunma hattı haline gelmiştir. Literatürdeki kapsamlı incelemeler, Büyük Dil Modellerinin (LLM) test hazırlığı ve program onarımı gibi görevlerde devrim niteliğinde potansiyeller sunduğunu göstermektedir [1]. Ancak, yaygın kullanımlarına rağmen SCA araçları "eyleme geçirilebilirlik açığı" (actionability gap) yaratan iki temel kısıtla karşı karşıyadır.

İlk kısıt, geliştiricilerde "uyarı yorgunluğuna" neden olan yüksek hatalı pozitif (false positive) oranlarıdır. Modern kod tabanlarının karmaşıklığı, geleneksel statik araçların hassasiyetini zorlamakta ve LLM tabanlı doğrudan anlamsal analiz yöntemlerine olan ihtiyacı artırmaktadır [2]. İkinci kısıt ise SCA uyarılarının geçici (ephemeral) doğasıdır; bu uyarılar potansiyel bir riski işaret etse de hatanın varlığına dair somut bir kanıt sunmamakta ve düzeltme sonrası kalıcı bir regresyon testi bırakmamaktadır. Bu durum özellikle erişilebilirlik [3] ve güvenlik konfigürasyonları [4] gibi, manuel doğrulanması hem maliyetli hem de hata payı yüksek olan kritik alanlarda güvenlik açıklarının üretime sızmasına yol açmaktadır.

Büyük Dil Modelleri, geleneksel analiz araçlarının kısıtlarını aşarak ham kod içinden derinlemesine güvenlik ve kalite çıkarımları yapma konusunda umut verici sonuçlar sergilemektedir [5]. Bu makale, SCA araçlarından bağımsız olarak ham kaynak kodundaki kusurları tespit eden ve bunları somut, yürütülebilir birim testlerine dönüştüren "Kusurdan-Teste" (Defect-to-Test - D2T) prensibi üzerine inşa edilen LLM-QA çerçevesini sunmaktadır. LLM-QA, analiz sürecini "The Funnel" (Huni) modeliyle daraltarak kaynak kodunu test

üretimi için teknik şartnameler olarak yorumlar. Sistemin merkezinde yer alan Yapısal Düşünce Zinciri (Structural Chain of Thought - SCoT) yöntemi, LLM'lerin karmaşık mantıksal akıl yürütme kapasitesini kullanarak kusurları yeniden üreten yüksek kaliteli test senaryoları oluşturmasını sağlar [6]. Yapay zeka tarafından üretilen kodların okunabilirlik ve dökümantasyon kalitesi açısından sunduğu avantajlar [7], LLM-QA'nın ürettiği "Kanıtlayıcı Testlerin" (Proving Test) sadece birer doğrulama aracı değil, aynı zamanda projenin test paketine eklenen sürdürülebilir varlıklar olmasını temin eder. Böylelikle yazılım kalite güvencesi, hataları bildiren pasif bir yapıdan, onları kalıcı bir güvenlik ağı (safety net) içinde kontrol altına alan aktif bir savunma mekanizmasına evrilir.

Bu çalışmanın devamında, öncelikle yazılım test süreçlerinde Büyük Dil Modellerinin (LLM) kullanımı ve statik analiz araçlarının kısıtları üzerine mevcut literatür incelenmektedir. İkinci bölümde, önerilen LLM-QA sisteminin mimari temellerini oluşturan "The Funnel" (Huni) stratejisi ve sistemin karar mekanizmasını yöneten dört aşamalı istem zinciri (Structure Map, Scout, Auditor, Fault Validator) detaylandırılmaktadır. Üçüncü bölümde, sistemin operasyonel entegrasyonu, Git operasyonları ve CI/CD süreçlerine dahil edilme süreci ele alınmaktadır. Son bölümde ise sistemin hata tespit hassasiyeti ve "eyleme geçirilebilir" test sentezleme yeteneği üzerine elde edilen bulgular tartışılmakta, çalışmanın sunduğu katkılar ve gelecek perspektifleri sunulmaktadır.

## II. Ilgili araştırmalar

[Benchmark Tablosu Ekle]

Bu bölüm, LLM-QA'nın ele almayı amaçladığı teknik boşluğu vurgulayarak; otomatik test üretimi, güvenlik doğrulaması ve Büyük Dil Modellerinin (LLM) Yazılım Mühendisliğindeki çağdaş uygulamalarını sentezlemektedir.

Otomatik test üretimi olgun bir araştırma alanı olmasına rağmen, "Test Kâhini" (Test Oracle) belirlemede—yani insan müdahalesi olmadan bir testin başarı veya başarısızlık kriterinin tanımlanmasında—önemli engeller devam etmektedir. Literatür taramaları, LLM'lerin birim test hazırlığı ve program onarımı gibi görevlerde geleneksel yöntemlerin ötesine geçtiğini göstermektedir [1]. Ancak, özellikle mutasyon testi gibi gelişmiş tekniklerle desteklenmediğinde, LLM tarafından üretilen testlerin hata yakalama etkinliği (test efficiency) üzerinde hala tartışmalar sürmektedir [9].

Geleneksel araçların kısıtları, özellikle Erişilebilirlik Testi gibi anlamsal yorumlama gerektiren alanlarda belirgindir. Mevcut otomatize araçların (Axe, Lighthouse vb.) WCAG yönergelerinin yalnızca kısıtlı bir kısmını kapsayabildiği ve bağlama dayalı hataları (context-dependent flaws) sıklıkla kaçırdığı kanıtlanmıştır [3]. Bu durum, kullanıcı arayüzünün (UI) sadece teknik yapısını değil, amacını da kavrayabilen akıllı sistemlere olan ihtiyacı doğurmaktadır. Benzer bir karmaşıklık performans testlerinde de görülmektedir. Mikro hizmet mimarilerinde kritik iş yüklerinin belirlenmesi, statik eşiklerin ötesinde nedensellik analizi gerektirmektedir. Mascia ve ark., geleneksel betiklerin yetersiz kaldığı "performans bütçesi" tanımlamalarında nedensellik-artırılmış (causality-enhanced) LLM'lerin kullanımını önermektedir [8].

Güvenlik testi, fonksiyonel doğrulamadan farklı olarak, bir saldırganın zihniyetini simüle eden yüksek bir kanıt yükü gerektirir. Erken dönem araştırmalar, etkili güvenlik değerlendirmesinin ancak sistemin sağlamlığını (robustness) test eden "hata enjeksiyonu" (fault injection) yöntemleriyle mümkün olduğunu ortaya koymuştur [10]. Güncel bulgular ise modern yazılım döngüsündeki risklerin evrildiğini göstermektedir; zira LLM tarafından üretilen kodların önemli bir kısmının (%45), karmaşık mantıksal hatalardan ziyade temel güvenlik kısıtlamalarının ihmal edilmesi nedeniyle zafiyet barındırdığı bildirilmiştir [14]. Bu durum, yapay zeka destekli kod üretim araçlarında (Copilot vb.) yüksek oranda CWE (Common Weakness Enumeration) açığı tespit eden ampirik çalışmalarla da desteklenmektedir [12].

En önemlisi, güvenlik kusurları sadece kod mantığıyla sınırlı değildir; geliştiricilerin CSP ve HSTS gibi "görünmez" güvenlik konfigürasyonlarını sıklıkla gözden kaçırdığı tespit edilmiştir [13]. LLM-QA, bu eksiklikleri basit "en iyi uygulama" tavsiyeleri olarak değil, yokluklarını dinamik olarak kanıtlayan başarısız test senaryoları (failing test cases) olarak ele alarak literatürdeki bu açığı kapatmaktadır.

Yapay zeka tarafından üretilen testlerin uzun vadeli bakım maliyeti (teknik borç) konusunda literatürde tereddütler bulunsa da, son ampirik çalışmalar yapay zeka çıktılarının okunabilirlik ve dökümantasyon kalitesi açısından insan yazımı kodlarla yarıştığını, hatta karmaşık görevlerde onları geçtiğini göstermektedir [7]. LLIFT [2] ve Abtahi & Azim'in [5] çalışmaları gibi güncel yaklaşımlar, statik analizi LLM ile zenginleştirerek hata tespitine odaklanmaktadır. Ancak bu sistemler, SCA araçlarının çıktılarına bağımlıdır ve genellikle tespit edilen hatayı dinamik bir "güvenlik ağına" dönüştürmeden doğrudan onarım aşamasına geçmektedir. LLM-QA, herhangi bir SCA aracına muhtaç olmaksızın kaynak kodunu doğrudan analiz edebilen huni (funnel) yapısı ve "Kusurdan-Teste" (Defect-to-Test) boru hattıyla ayrışır. Bu sayede her bulgu, kalıcı ve yürütülebilir bir doğrulama yapıtına dönüşür.

## III. Metodoloji

LLM-QA sistemi, geleneksel statik analiz (SCA) araçlarının ürettiği geçici uyarıları, projenin kod tabanına entegre edilen kalıcı ve yürütülebilir "güvenlik ağlarına" (safety nets) dönüştüren uçtan uca bir "Kusurdan-Teste" (Defect-to-Test - D2T) boru hattıdır. Sistem, ölçeklenebilirliği sağlamak adına durum bilgisiz (stateless) bir mimari üzerine inşa edilmiştir ve devasa kod tabanlarını tek bir kanıtlanabilir test dosyasına daraltmak üzere "The Funnel" (Huni) stratejisini kullanır.

### A. Sistem Mimarisi ve Operasyonel Veri Akışı

Sistemin operasyonel akışı, token maliyetini optimize eden ve doğruluğu artıran beş ana adımdan oluşmaktadır:

1. GitHub İletişimi: Kullanıcı tarafından sağlanan URL üzerinden GitHub REST API aracılığıyla depo yapısı hiyerarşik bir dizin ağacı olarak alınır.

2. Local Ignore: .llmqaignore dosyası aracılığıyla node_modules/, dist/ gibi deterministik olarak

| Aşama | Rol | Hedef Model | Çıktı Şeması | Temel Görev |
|---|---|---|---|---|
| Eliminator | Senior Build Engineer | Gemini 2.5 Flash Lite | ignore_list[] | Dosya ağacı temizliği ve maliyet optimizasyonu |
| Investigator | QA Automation Lead | Gemini 2.5 Flash | category, reason | Toplu dosya işleme ve risk sınıflandırması |
| Identifier | Principal Engineer | Gemini 2.5 Flash | findings[{trigger}] | Satır bazlı audit ve bağlam döngüsü |
| Creator | Senior SDET | Gemini 3 Flash | code: string | Yürütülebilir test sentezi ve sandbox doğrulaması |

Tablo 1: Dört Aşamalı İstem Zinciri Karşılaştırmalı Özeti

filtrelenebilecek dizinler ayıklanarak LLM maliyeti henüz süreç başlamadan minimize edilir.

3. LLM Boru Hattı: Kalan dosyalar, kademeli bir eleme mekanizması olan huni yapısına dahil edilir.

4. Test Sentezi: Tanımlanan her kusur için projenin teknoloji yığınına uygun (şimdilik sadece Python) özgün test senaryoları oluşturulur.

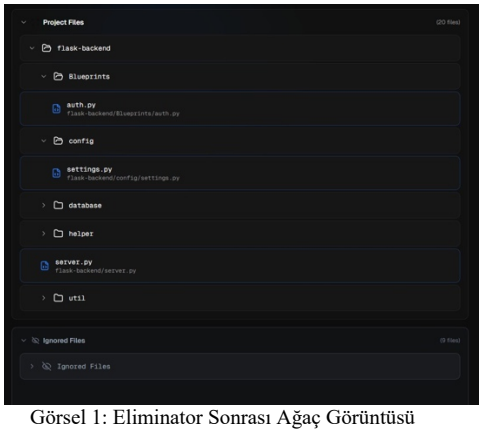
B. Dört Aşamalı LLM İstem (Prompt) Zinciri

LLM-QA sisteminin temel muhakeme gücü, token verimliliğini optimize eden ve yanlış pozitifleri (false positives) kademeli olarak azaltan, birbirini besleyen dört aşamalı bir istem zincirine dayanmaktadır. Her aşama, karmaşık bir "Huni" (Funnel) stratejisinin parçası olarak, veriyi bir sonraki aşama için daha rafine ve odaklanmış bir hale getirir.

1) Aşama I - ELIMINATOR (Yapı Haritalama): Bu aşamada, maliyet odaklı Gemini 2.5 Flash Lite modeli bir "Kıdemli Yapılandırma ve Yayın Mühendisi" (Senior Build & Release Engineer) rolünü üstlenir.

• İstem Şeması: Sistem, projenin dosya ağacını analiz ederek examples/, dist/, test/ gibi hata barındırma potansiyeli düşük dizinleri ve .md, .css, .html gibi yürütülebilir mantık içermeyen uzantıları belirlemek üzere tasarlanmış bir System Instruction kullanır.

• Çıktı Yapısı: Yanıt şeması, sistemin deterministik olarak filtreleme yapabilmesi için bir ignore_list dizisi ve elenen öğelerin sayısını (count) içeren yapılandırılmış bir JSON nesnesidir.

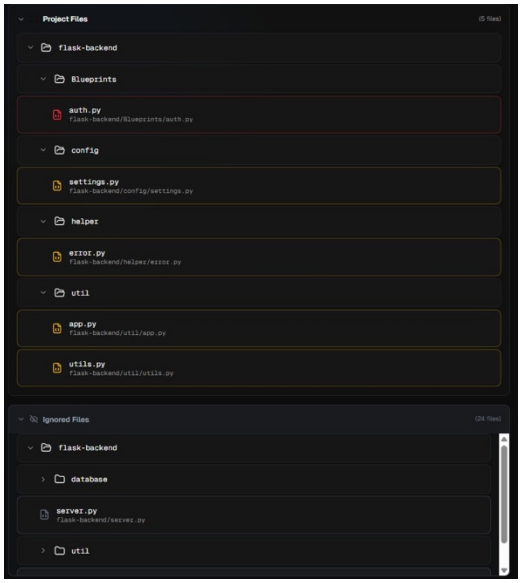



Görsel 1: Eliminator Sonrası Ağaç Görüntüsü

2) Aşama II - INVESTIGATOR (Toplu Risk Sınıflandırması): Bir önceki aşamadan geçen "mavi" (kritik) dosyalar, Gemini 2.5 Flash modeli tarafından bir "QA Otomasyon Lideri" rolüyle taranır.

• İstem Şeması: Dosyalar, Google File API aracılığıyla 3-5 dosyalık partiler (batch) halinde sisteme yüklenir. User Template, her dosyanın içerik risk profilinin çıkarılmasını talep eder.

• Sınıflandırma: Dosyalar; ignore (mantık içermeyen), suspect (şüpheli/karmaşık) ve error (açık kusurlu) olarak etiketlenir. Her etiketleme, sistemin sonraki aşamada odaklanacağı "gerekçe" (reason) verisiyle desteklenir.

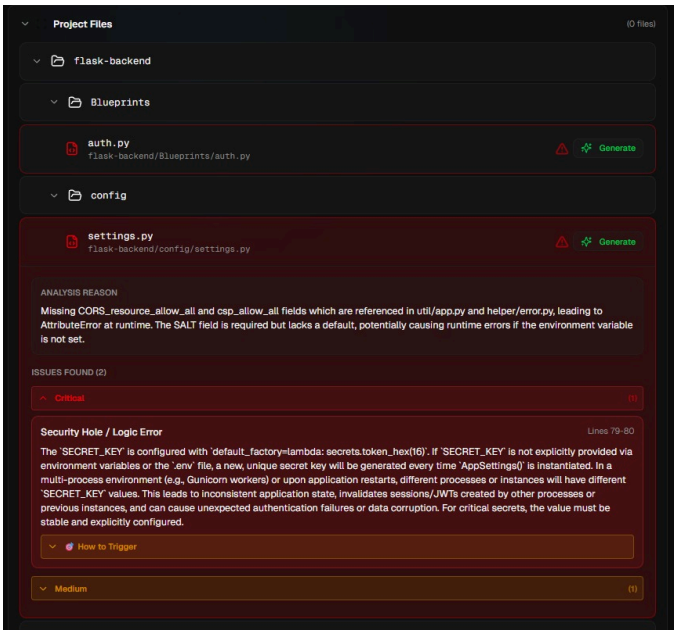



Görsel 2: Investigator Sonrası Ağaç Görüntüsü

3) Aşama III - IDENTIFIER (Derinlemesine Hata Tanımlama): Şüpheli olarak işaretlenen dosyalar, bir "Kıdemli Baş Mühendis" (Senior Principal Engineer) rolündeki model tarafından satır satır denetlenir.

• İstem Şeması: System Instruction, yarış durumları (race conditions), sınır hataları (off-by-one) ve güvenlik açıklarını tespit etmeye odaklanır. Eğer analiz sırasında eksik bir bağımlılık (dependency) fark edilirse, "Context Loop" (Bağlam Döngüsü) mekanizmasıyla ilgili dosyalar otomatik olarak isteme dahil edilir.

• Bulgu Sentezi: Çıktı, line_range, severity, type, description ve en önemlisi hatayı tetikleyecek senaryoyu tanımlayan trigger alanlarını içeren kesin bir JSON veri setidir.



Görsel 3: Identifier Sonrası Ağaç Görüntüsü

4) Aşama IV - CREATOR (Kanıtlanmış Test Sentezi): Son aşamada, bir "Kıdemli Yazılım Test Geliştirme Mühendisi" (Senior SDET), tanımlanan kusuru somut bir Python testine dönüştürür.

• İstem Şeması: System Instruction, harici bağımlılıkların (veritabanı, ağ) mock'lanmasını ve testin tamamen bağımsız (self-contained) olmasını şart koşar.

• Doğrulama ve İterasyon: Model, projenin teknoloji yığınını (package.json, pom.xml vb.) tespit eder. Sentezlenen kod, LLM'in dahili kod yürütme (sandbox) ortamında çalıştırılır; sözdizimi veya çalışma zamanı hatası alınması durumunda sistem, hatasız ve yürütülebilir bir "Kanıtlayıcı Test" (Proving Test) üretilene kadar iteratif düzeltme yapar.

### C. Token Optimizasyonu ve Maliyet Yönetimi

Huni mimarisi, tüm dosyaları doğrudan derin analize göndermek yerine kademeli daraltma yaparak token maliyetini yaklaşık 15 kat azaltır. 500 dosyalık bir projede huni yapısı sayesinde ~2.000.000 token yerine yalnızca ~135.000 token harcanarak yüksek verimlilik sağlanır. Ayrıca, basit görevlerde ucuz modellerin, karmaşık sentezlerde ise güçlü modellerin kullanıldığı model stratifikasyonu uygulanır.

### D. Durum Bilgisiz (Stateless) API Tasarımı

Sistem, ölçeklenebilirliği garanti altına almak adına FastAPI üzerinde tasarlanmıştır. Oturum yönetimi benzersiz session_id'ler ile yapılırken, kullanıcı arayüzü ile iletişim Server-Sent Events (SSE) üzerinden gerçek zamanlı olarak sağlanır. Analiz bitiminde geçici dosyalar otomatik olarak temizlenerek veri güvenliği ve sistem hafifliği korunur.

## SONUÇ

Bu çalışma, geleneksel Statik Kod Analizi (SCA) araçlarının yarattığı "eyleme geçirilebilirlik açığını" (actionability gap) kapatmak amacıyla geliştirilen, Büyük Dil Modeli (LLM) tabanlı LLM-QA çerçevesini sunmuştur,. Önerilen sistem, ham kaynak kodunu doğrudan analiz ederek karmaşık mantık kusurlarını ve güvenlik açıklarını yürütülebilir "Kanıtlayıcı Testlere" (Proving Tests) dönüştüren uçtan uca bir "Kusurdan-Teste" (D2T) boru hattı sağlamaktadır.

Çalışma kapsamında elde edilen temel bulgular ve sistemin sağladığı katkılar şu şekilde özetlenebilir:

• Operasyonel Verimlilik: "The Funnel" (Huni) stratejisi ve dört aşamalı istem zinciri sayesinde, sistem devasa kod tabanlarını anlamsal bir süzgeçten geçirerek analiz maliyetini yaklaşık 15 kat azaltmayı başarmıştır,. 500 dosyalık bir projede huni mimarisi kullanımı, token tüketimini 2.000.000 seviyesinden ~135.000 seviyesine indirerek ölçeklenebilir bir model sunmuştur.

• Güvenlik Ağlarının Oluşturulması: Geleneksel araçların aksine LLM-QA, tespit edilen kusurları geçici uyarılar olarak bırakmamakta; bunları projenin test paketine dahil edilebilir ve CI/CD süreçlerinde koşturulabilir, sürdürülebilir varlıklara dönüştürmektedir.

• Hata Doğrulama Hassasiyeti: Sistem, özellikle manuel doğrulanması maliyetli olan yarış durumları (race conditions) ve görünmez güvenlik konfigürasyonu eksikliklerini, "başarısız olan test senaryoları" (failing test cases) üzerinden dinamik olarak ispatlamaktadır.

• İteratif Gelişim: CREATOR aşamasında kullanılan dahili kod yürütme (sandbox) mekanizması, LLM tarafından üretilen testlerin sözdizimi ve mantık hatalarından arındırılmasını sağlayarak yalnızca doğrulanmış kodların sisteme dahil edilmesini garanti altına almıştır,.

Sonuç olarak LLM-QA, yazılım kalite güvence süreçlerini hataları sadece bildiren pasif bir yapıdan, onları kalıcı bir "güvenlik ağı" (safety net) içinde kontrol altına alan aktif ve kanıta dayalı bir savunma mekanizmasına evrimleştirmiştir,. Gelecekteki çalışmalar, sistemin desteklediği teknoloji yığınlarının Python ötesine genişletilmesi ve daha karmaşık mikro hizmet mimarilerindeki nedensellik analizlerinin derinleştirilmesi üzerine odaklanacaktır,.

## TEŞEKKÜR

Bu makalenin düzenlenmesi ve doğrulanmasında bize yönlendiren Muhammed Baykara hocamıza teşekkür ederiz.

## REFERENCES

[1] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing With Large Language Models: Survey, Landscape, and Vision," IEEE Transactions on Software Engineering, vol. 50, no. 4, pp. 911-936, 2024.

[2] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2024.

[3] T. Kodithuwakku and D. Wickramaarachchi, "Assessing the Limits of Automated Accessibility Testing: Insights for QA Engineers and Tool Developers," in 2025 5th International Conference on Advanced Research in Computing (ICARC), IEEE, 2025.

[4] S. Pargaonkar, "Advancements in Security Testing: A Comprehensive Review of Methodologies and Emerging Trends in Software Quality Engineering," International Journal of Science and Research (IJSR), vol. 13, no. 8, 2023.

[5] M. F. Kharma, S. Choi, M. Alkhanafseh, and D. Mohaisen, "Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis," arXiv preprint arXiv:2502.01853, 2025.

[6] D. Schwachhofer et al., "Large Language Model-based Optimization for System-Level Test Program Generation," in 2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), IEEE, 2024.

[7] H. Fawareh, H. M. Al-Shdaifat, and M. Khouj, "Investigates the impact of AI-generated code tools on software readability code quality factor," in 2024 25th International Arab Conference on Information Technology (ACIT), IEEE, 2024.

[8] C. Mascia et al., "Microservices Performance Testing with Causality-enhanced Large Language Models," in 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), 2025.

[9] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained Large Language Models and mutation testing," Information and Software Technology, vol. 171, 107468, 2024.

[10] R. Kaksonen, M. Laakso, and A. Takanen, "Software security assessment through specification mutations and fault injection," in Communications and Multimedia Security Issues of the New Century, Kluwer Academic Publishers, 2001, pp. 2704–2715.

[11] A. S. Al-Ghamdi, "A Survey on Software Security Testing Techniques," International Journal of Computer Science and Telecommunications, vol. 4, no. 4, 2013.

[12] Y. Fu et al., "Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study," ACM Transactions on Software Engineering and Methodology, vol. 34, no. 8, 2025.

[13] N. Perry, V. P. Kemerlis, and B. Rodes, "The invisible risk: Security configuration omissions in modern web applications," arXiv preprint arXiv:2408.01234, 2024.

[14] M. F. Kharma, S. Choi, M. Alkhanafseh, and D. Mohaisen, "Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis," arXiv preprint arXiv:2502.01853, 2025