ChatGPT

Stacks Blockchain Smart Contract Monitoring Backend – Kapsamlı Analiz ve Yol Haritası

Sistem Mimarisi ve Bileşenlerinin Analizi

Proje Tanımı: Stacks Chain Monitor sistemi, Stacks blokzincirindeki akıllı kontrat işlemlerini gerçek zamanlı izleyip önceden tanımlı kurallara göre uyarılar üreten bir **izleme ve bildirim platformudur** [1]. Örneğin belirli bir kontratta büyük miktarda token transferi gerçekleştiğinde veya bir işlem başarısız olduğunda, sistem bunu tespit edip kullanıcıya e-posta veya webhook ile anında bildirir [2]. Bu sayede blockchain üzerindeki kritik olaylar anlık yakalanıp kullanıcılar proaktif bilgilendirilir.

Genel İşleyiş: Sistem, **Chainhook** adlı Stacks blockchain'e özgü bir mekanizma sayesinde çalışır. Stacks ağında yeni bir blok üretildiğinde veya ilgili bir olay gerçekleştiğinde Chainhook, olayı JSON formatında backend'e gönderir [3]. Backend tarafında tanımlı bir HTTP webhook endpoint'i (`/api/v1/webhook/chainhook`) bu çağrıları alır ve işleme başlar. Gelen ham blockchain verisi (blok, işlemler ve olaylar) öncelikle domain modelindeki nesnelere pars edilir, ardından sırasıyla:

- **Veritabanı Kaydı:** Yeni blok ve içerdiği işlemler, `StacksBlock` ve `StacksTransaction` entity'leri olarak PostgreSQL veritabanına kaydedilir. Her bir işlem içindeki alt olaylar (token transferi, kontrat çağrısı vb.), polimorfik yapıda tanımlanan `TransactionEvent` entity'si olarak ilişkilendirilmiş bir şekilde veritabanına eklenir [4] [5]. Bu sayede blokzincirde gerçekleşen her bir düşük seviye olay sistem içinde kendi tablolarında kayıt altına alınmış olur. Domain modelinde **StacksBlock** ile **StacksTransaction** arasında bire-çok ilişki, **StacksTransaction** ile **TransactionEvent** arasında yine bire-çok ilişki bulunmaktadır (bir blok birçok işlem içerir, her işlem birçok alt olaya sahip olabilir) [5] [6]. İlaveten, her `TransactionEvent` olayının alt türüne göre (token transferi, akıllı kontrat çağrısı, vb.) alt entity sınıfları mevcuttur; mevcut sistemde bunlar **join inheritance** ile tek tabloda tutulsa da alt sınıflar ile temsil edilir [7] (ör. `FTTransferEvent`, `ContractCall`, `ContractDeployment` gibi).

- **Kural Değerlendirme:** Yeni eklenen her **TransactionEvent** için sistem derhal tanımlı **AlertRule**'lar ile karşılaştırma yapar [8]. **AlertRule** entity'si kullanıcıların tanımladığı esnek kuralları temsil eder; örneğin "şu kontratta çağrılan fonksiyon X ise ve gönderilen miktar > 1000 ise uyarı gönder" gibi bir kural JSON formatında bu entity'de saklanır [9]. Sistemde **5 farklı tipte uyarı kuralı** desteklenmektedir: `CONTRACT_CALL`, `TOKEN_TRANSFER`, `FAILED_TRANSACTION`, `PRINT_EVENT` (kontrat içinden yapılan `print` logları için) ve `ADDRESS_ACTIVITY` (belirli bir cüzdan adresinin aktiviteleri) [10] [11]. Her AlertRule bir **kullanıcı**ya ve opsiyonel olarak izlenen bir kontrata (`MonitoredContract`) bağlıdır [12] [13]. Sistem bir olayı değerlendirirken ilgili TransactionEvent'in detaylarını tüm aktif kuralların koşullarıyla eşleştirir. Mevcut implementasyonda bu eşleştirme *cache'lenmiş* aktif kurallar üzerinden yapılır, ancak yine de potansiyel olarak **O(n)** karmaşıklığında bir döngü içermektedir (bu tasarımın performansla ilgili etkileri aşağıda ele alınacaktır) [14] [15].

- **Bildirim Oluşturma:** Bir TransactionEvent belirli bir AlertRule'un koşullarını sağlıyorsa ve kural o an tetiklenebilir durumdaysa (ör. cooldown süresinde değilse), sistem yeni bir **AlertNotification** kaydı oluşturur [16]. AlertNotification, tetiklenen kuralı, ilişkili işlemi/olayı ve gönderim kanalını

(E-POSTA veya WEBHOOK) içerir [17] [18]. Bu nesne veritabanında "PENDING" durumu ile kaydedilir.

- **Bildirim Gönderme:** Ardından bir **NotificationDispatcher** servisi devreye girerek bekleyen bildirimi uygun kanalla göndermeye çalışır. Sistem e-posta gönderimleri için **EmailNotificationService**, webhook gönderimleri için **WebhookNotificationService** tanımlarına sahiptir – her ikisi de ortak bir arayüzü (muhtemelen NotificationService) implement edip `send()` ve `supports()` metotları içerir [19] [20]. Gönderim denemesi başarılı olursa AlertNotification "SENT" olarak işaretlenir, başarısız olursa "FAILED" durumuna alınır ve **otomatik yeniden deneme (retry)** mekanizması tetiklenir [21]. Mevcut kodda bildirim gönderimi ayrı bir thread'de @Async ile yapılmakta ancak retry/circuit-breaker mekanizması henüz manuel olarak kodlanmamıştır (aşağıda detaylandırılacak) [22] [23]. Başarılı gönderimlerde ilgili zaman damgaları kaydedilir, başarısızlıklarda ise sistem bir süre bekleyip yeniden göndermeyi dener (örn. e-posta servisi geçici olarak down ise) [24] [25].

Katmanlı Mimari: Proje, **Clean Architecture** prensiplerine uygun olarak dört katmana ayrılmıştır: **Sunum (Presentation)**, **Uygulama (Application)**, **Domain (Etki Alanı)** ve **Altyapı (Infrastructure)** [26]. Her katman kendi sorumluluğuna odaklanır ve bir alt katmana bağımlıdır, üst katmanlardan bağımsızdır – bu, Clean Architecture'ın bağımlılık kuralına uygundur (iç katmanlar dış katmanlardan habersizdir) [27] [28]. Özetle:

- **Sunum Katmanı:** Dış dünya ile sistemin buluşma noktası olan REST API arabirimini içerir. Spring Boot *@RestController*'lar bu katmandadır (örn. `WebhookController`, `AlertRuleController`, `TransactionQueryController` vb.). Bu katman gelen HTTP isteklerini alır, doğrulamalarını yapar ve uygun DTO'lar ile uygulama katmanına iletir; sonuçları da HTTP response olarak dışarı döndürür [29] [30]. Örneğin `WebhookController` POST `/api/v1/webhook/chainhook` endpoint'i Chainhook'tan gelen çağrıları alırken, `AlertRuleController` kullanıcıların kural tanımlamalarını yönetmesine yarar. İstisna yönetimi için global bir `GlobalExceptionHandler` (@ControllerAdvice) tanımlıdır, validasyon için DTO seviyesinde @Valid anotasyonları kullanılır [30] [31].

- **Uygulama Katmanı:** İş kurallarının orkestrasyonundan sorumlu servisler ve use-case sınıflarını barındırır. **Controller**'lardan gelen çağrılar önce buradaki **Service/UseCase** sınıflarına iletilir [32]. Her önemli iş akışı için ayrı bir servis bulunmaktadır. Örneğin `ProcessChainhookPayloadUseCase` sınıfı, Chainhook'tan gelen bir yeni blok payload'ını alıp işleme adımlarını yürütür (block ve transactions veritabanına kaydet, olayları çıkar, kuralları değerlendir) [33]. Bu katmanda genelde metodlar @Transactional ile işaretlenerek bir iş akışı boyunca atomiklik sağlanır [34]. Örneğin yeni bir blok geldiğinde, UseCase içinde `StacksBlockRepository` aracılığıyla blok tablo kaydı, ardından her işlem için `StacksTransactionRepository` kaydı yapılır, sonra her işlem içindeki event'ler `TransactionEventRepository` ile eklenir [35]. Uygulama katmanı, domain nesnelerini kullanarak iş adımlarını yönetir ve gerekirse farklı servisler arası koordinasyon sağlar [36]. Mevcut projede `AlertMatchingService` (gelen işlemleri kurallarla eşleştiren), `AlertRuleService` (kural yönetimi CRUD), `AuthenticationService` (register/login işlemleri) gibi bileşenler bu katmandadır [37] [38]. Kod yapısı tek sorumluluk prensibini gözetir, her servis belirli bir iş yükünü üstlenir (SRP prensibi) [39].

- **Domain Katmanı:** İş problemini yansıtan **varlık (entity) modelleri** ve **iş kuralları** burada yer alır. Domain katmanı, uygulamanın merkezidir ve *"rich domain model"* anlayışıyla tasarlanmıştır: Örneğin **AlertRule** entity'si içinde `canTrigger()` veya `isInCooldown()` gibi metodlar

barındırır ve kendi tetiklenme koşullarını kendi üzerinde hesaplar [40] [41]. Bu sayede iş mantığı büyük ölçüde domain nesneleri üzerinde toplanmıştır (DDD ilkelerine uygun şekilde). Domain katmanında **Repository** arayüzleri de tanımlıdır ancak bunlar sadece bir arabirim olarak kalır; gerçek veri erişimi altyapı katmanında sağlanır [42] [43]. Domain modelinde bu projeye özgü kavramları temsil eden 9 temel entity tanımlanmıştır [44] [45]: Başlıcaları **StacksBlock**, **StacksTransaction**, **TransactionEvent**, **ContractCall**, **ContractDeployment**, **MonitoredContract**, **AlertRule**, **AlertNotification** ve **User** entity'leridir. Bu model, Stacks blokzincirinden gelen tüm veriyi kapsayacak şekilde tasarlanmıştır [46] [47]. Örneğin **StacksTransaction** bir işlemin tüm detaylarını (gönderen, sponsor, ücret, nonce, başarı durumu, ham sonuç vs.) tutacak alanlara sahipken [48] [49], **TransactionEvent** tekil bir blockchain olayını (token transfer, kontrat çağrısı, vs.) temsil eder ve *eventType* alanına göre alt sınıflara ayrılır [6]. **AlertRule** tek bir uyarı kuralını temsil eder; JSON formatlı koşul tanımı, kurala ait son tetiklenme zamanı, ilgili kullanıcı ve kontrat bilgileri gibi alanları içerir [50] [51]. **User** entity'si sistem kullanıcılarını temsil eder (email, passwordHash, roller vb.) [52]. Domain katmanı, **uygulamanın ne yaptığı** ile ilgilenir, *nasıl yaptığı* ile değil – örneğin bir AlertNotification'in nasıl e-posta yolladığı bilgisi domain'de olmaz, sadece "EMAIL kanalıyla gönderilecek" bilgisi domain'de tutulur.

- **Altyapı Katmanı:** Uygulamanın dış dünya ile iletişimi ve teknik detayları barındırır. Veritabanı erişimi (Spring Data JPA repository implementasyonları), e-posta gönderimi, harici servis entegrasyonları, background job'lar, konfigürasyonlar bu katmandadır [53] [54]. Mevcut projede `infrastructure.config` paketi altında güvenlik konfigürasyonu ve filtreleri (JWT, HMAC, RateLimit filtreleri vb.), e-posta gönderimi için `NotificationConfig`, Chainhook JSON payload'ını parse eden `ChainhookPayloadParser` gibi bileşenler bulunmaktadır [55] [56]. Ayrıca Spring Boot Starter'ları (Mail, Cache, Security, etc.) aracılığıyla altyapı servisleri konfigure edilir. Örneğin **CustomUserDetailsService** domain'deki User verisini Spring Security ile entegre ederken [57], **JwtTokenService** JWT üretim ve doğrulama fonksiyonlarını kapsüller [56]. Repository implementasyonları ise Spring Data JPA tarafından otomatik olarak sağlanır (`AlertRuleRepository`, `StacksTransactionRepository` gibi arayüzler `JpaRepository` extend ederek altyapı katmanında somutlanır) [42]. Altyapı katmanı ayrıca uygulamanın çıkış yapacağı sınır noktalarını (e-posta servisi, harici webhook çağrısı vb.) içerdiğinden, bu katmandaki bileşenlerin **dış sistemlerle iletişimde güvenlik** ve **hata toleransı** sağlaması önemlidir.

Teknoloji Yığını: Proje Java 17 ve Spring Boot 3.x üzerine inşa edilmiştir [58]. Önemli bağımlılıklar arasında Spring Web, Spring Data JPA, Spring Security, Spring Validation, Spring Mail, Spring Cache, Spring Actuator, Postgresql sürücüsü, FlywayDB, Micrometer/Prometheus ve test için Testcontainers yer alıyor [59] [60]. Bu stack, modern bir kurumsal Java uygulaması için gereken tüm temel bileşenleri sağlamaktadır. Veritabanı olarak PostgreSQL kullanılmış ve Flyway ile şema versiyonlaması uygulanmıştır (ör. `V1__initial_schema.sql` ile temel tablo yapıları oluşturulmuş) [61] [62]. Spring Boot Actuator ve Micrometer Prometheus registry bağımlılıkları, uygulamanın **gözetlenebilirlik (observability)** özelliklerini (sağlık end-point'leri, metrikler) devreye almak için eklenmiştir.

Kod Organizasyonu: Kod yapısı fonksiyonlarına göre mantıksal paketlere ayrılarak düzenlenmiştir [63]. Örneğin:

- `api.controller` paketi altında tüm REST Controller sınıfları;
- `api.dto` altında dış dünya ile veri alışverişinde kullanılan DTO sınıfları;
- `domain.model` altında yukarıda bahsedilen tüm **entity** sınıfları (alt alanlara göre `blockchain`, `monitoring`, `user` gibi alt paketlerle gruplanmış);
- `domain.repository` altında JPA repository arayüzleri;

- `application.service` ve `application.usecase` altında uygulama katmanı servis ve use-case sınıfları;
- `infrastructure` paketi altında ise yapılandırma (security config vb.), harici servis adaptörleri (ör. e-posta gönderimi) ve yardımcı bileşenler yer almaktadır [64] [65].

Bu paketleme yaklaşımı sayesinde proje büyüdükçe bile okunabilirlik ve **bakım kolaylığı** korunur – bir geliştirmeci örneğin veri modeliyle ilgili bir değişiklik yapmak istediğinde `domain.model` paketine bakarken, bir API endpoint'ini değiştirmek istediğinde `api.controller` altında ilgili sınıfı bulacağını bilir [66] [67]. Her katman ve paket, sorumlu olduğu işlevleri üstlenir ve katmanlar arası iletişim arayüzler üzerinden gerçekleşir [68] [69]. Bu sayede sisteme yeni özellik eklemek oldukça kolaylaşır; örneğin yeni bir alert kural tipi ekleneceği zaman domain'e yeni bir AlertRule alt sınıfı ve ilgili kontrol/servis kodu eklenerek diğer kısımlarla minimum entegre olacak şekilde gelişim sağlanabilir [69].

Kritik İşlevler ve Bileşenler: Sistemin ana akışı dışında dikkat çeken bazı kritik bileşenler şunlardır:

- **ChainhookPayloadParser & ProcessChainhookPayloadUseCase:** Gelen ham JSON payload'ın domain nesnelerine dönüştürülmesi işini yapar. JSON verisi içinde 50+ işlem bile olabilir; bunlar parçalara ayrılıp Transaction nesnelerine ve alt event nesnelerine çevrilir. Parse işlemi polimorfik DTO'lar ve Jackson alt tip anotasyonları (@JsonSubTypes) ile sağlanmış durumda; 11 farklı event tipini tek bir payload içinde ayırt edebilen esnek bir DTO hiyerarşisi kurulmuştur [70] [71]. Ardından UseCase sınıfı, her bir blok payload'ını veritabanına kayıt ve kural değerlendirme adımlarını yönetir. Bu işlem yoğun olduğundan, **yüksek hacimli veride bellek yönetimi** ve **işlem süresi** kritik hale gelmektedir (bununla ilgili öneriler performans bölümünde ele alınacak).

- **AlertMatchingService & Rule Engine:** Her yeni transaction event geldiğinde çağrılan bu servis, ilgili transaction içindeki tüm alt olayları ve belki transaction'ın kendisini alarak bunları aktif kurallarla eşleştirir. Şu anki tasarımda AlertMatchingService, aktif AlertRule listesini bellek içi önbellekte tutar ve `evaluateTransaction()` metodu ile tek tek kuralları dolaşarak eşleşme kontrolü yapar [37] [14]. Bu kural motoru basit bir if/else kontrol mekanizmasıyla çalışmaktadır. **Cooldown** mekanizması da AlertRule seviyesinde uygulanmıştır: Her kuralın en son tetiklenme zamanı `lastTriggeredAt` alanında tutulur, eğer son tetiklemeden bu yana yeterli süre geçmediyse kural *tetiklenemez* durumda sayılır (böylece spam/tekrar uyarı engellenir). Bu kontrol AlertRule modelinin `canTrigger()` metodu içinde yapılmaktadır (örnek olarak `AlertRule.isInCooldown()` domain'de gerçeklenmiştir) [40] [41]. Kural eşleştirme işleminin performansı doğrudan sistemin ölçeklenebilirliğini etkilediğinden, bu bileşenin iyileştirilmesi önemli bir konudur (aşağıda ayrıntılı incelenecek).

- **NotificationDispatcher & NotificationService'ler:** Uyarı bildirimlerini kullanıcılara iletmekle sorumlu olan bileşendir. `NotificationDispatcher.dispatch()` metodu, veritabanında PENDING durumda bekleyen Notification kayıtlarını alıp uygun kanalla gönderir [72]. Kanal bağımlı detaylar (e-posta atmak veya webhook çağırmak) strateji deseni benzeri bir yapıyla çözülmüştür: Email ve Webhook için ayrı servisler vardır ve her biri kendi `send()` metoduyla implementasyon detayını kapsar [19] [73]. Dispatcher bu servisleri dinamik belirlemek için muhtemelen `supports(channel)` metodunu kullanır. Ayrıca belli aralıklarla başarısız bildirimleri yeniden denemek için `retryFailedNotifications()` metodu öngörülmüştür [72]. Şu anki implementasyonda bildirim gönderimi *asenkron* yapılmıştır (@Async anotasyonu ile ayrı bir thread pool'da çalışır) [22]. Ancak sistemde **geri kazanım (retry)** ve **devre kesici (circuit breaker)** mekanizmaları tam uygulanmadığından, örneğin bir webhook hedefi kalıcı olarak başarısız olursa bildirim kaybolabilir. Bu noktada bir **Dead Letter Queue** (DLQ) veya başarısız bildirimleri kuyruklayıp daha sonra manuel müdahale ile yeniden gönderebilme altyapısı eksiktir (bu da aşağıda ele alınacak önemli bir iyileştirme konusudur) [74] [25].

- **Security (JWT & HMAC):** Sistem, kullanıcı yönetimi ve kimlik doğrulama için temel hazırlıklara sahip ancak tam entegre değil. Örneğin bir **AuthenticationController** ve **AuthenticationService** mevcut; bunlar `register` ve `login` endpoint'leri sunuyor [75] [38]. Domain'de **User** entity'si ve muhtemelen sabit bir `USER` rolü tanımlı [76]. Spring Security konfigürasyonu kısmen hazır: `SecurityConfiguration` sınıfı ile WebSecurity konfigürasyonu yapılmış, muhtemelen JWT tabanlı auth filter'ı eklenmiş (JwtAuthenticationFilter mevcut görünüyor) [77] [56]. Ancak dokümantasyonda belirtildiği üzere JWT doğrulama tam devreye alınmamış; muhtemelen tüm endpoint'ler henüz JWT korumasına tabi değil veya token üretimi gibi detaylar eksik [78]. Öte yandan, Chainhook'tan gelen webhook çağrılarının güvenliği için HMAC imza kontrolü amacıyla bir `ChainhookHmacFilter` tanımlanmış durumda [79]. Bu filtre, Chainhook gönderilerinin header'ındaki imzayı bilinen bir secret ile karşılaştırarak doğrulama yapmayı hedefliyor. Nitekim `ChainhookSignatureValidator` gibi bir bileşen snippet'ı öneride mevcut: payload ve secret kullanarak HMAC-SHA256 hash hesaplıyor ve gelen imza ile kıyaslıyor [80]. Bu sayede blockchain verisi yalnızca yetkili kaynak (Chainhook) tarafından gönderildiğinde kabul edilecek şekilde güvence altına alınabiliyor. Ancak şu anki yapıtaşları tam olarak bağlanmamış olabilir; örneğin SecurityConfig içinde Chainhook endpoint'ine özel izin verilirken imza kontrol filter'ının eklendiğinden emin olunmalı (kodu incelediğimizde, "`/api/v1/webhook/chainhook`" endpoint'inin **permitAll** yapıldığı ancak HMAC kontrol eklenmesi gerektiği not edilmiştir [81] [82]).

- **Monitoring & Observability:** Projede Spring Boot Actuator ve Micrometer Prometheus entegrasyonu dahil edilmiş olsa da, özel metrik tanımları veya dağıtık izleme konfigürasyonları henüz eklenmemiştir. Actuator ile uygulama sağlığı, temel metrikler, belki önbellek istatistikleri gibi veriler alınabilir (nitekim `MonitoringController` altında `/api/v1/monitoring/stats/*` endpoint'leri tanımlı görünüyor) [83]. Ancak **uygulamaya özel** metrikler (ör. kaç blockchain olayı işlendi, bir payload'ın işlenme süresi, aktif kural sayısı gibi) için ekleme yapılmamıştır [84] [85]. Aynı şekilde loglama stratejisi veya dağıtık iz sürme (tracing) araçları entegrasyonu (Zipkin/Jaeger) henüz yoktur. Mevcut loglar büyük olasılıkla standart konsol çıktı formatında olup, JSON formatlı yapılandırılmış loglama veya merkezi log takibi (ELK stack gibi) planlanmamıştır. Observability konusundaki bu eksikler, production ortamda sorun giderme ve performans izleme açısından geliştirilmelidir (aşağıda öneriler bölümünde ayrıntılı değineceğiz).

Özetle, mevcut sistem mimarisi ve kod temelleri, **modern kurumsal Java standartlarıyla** büyük ölçüde uyumludur. Clean Architecture ve Domain-Driven Design prensipleri benimsenmiş olup, katmanlar arası ayrışma net bir şekilde uygulanmıştır [86] [87]. Domain modelleri Stacks blockchain işlemlerini ve izleme ihtiyaçlarını kapsayacak şekilde kapsamlı tasarlanmış, veri modelinde önemli performans noktaları (ör. TransactionEvent üzerinde 8 adet indeks, ContractCall üzerinde 5 indeks gibi) düşünülmüştür [88] [89]. Bu mimari sayede sistem, esnek ve genişlemeye müsait bir temel üzerinde kuruludur – yeni bir özellik eklemek istendiğinde ilgili katmana eklenerek, diğer katmanlarla minimum etkileşim olacak şekilde geliştirilebilir [69] [90]. Ancak, bir sonraki bölümde ele alacağımız üzere, **production-ready** (üretim ortamına hazır) hale gelebilmesi için bazı eksik noktaların giderilmesi ve endüstri standartlarına tam uyumun sağlanması gerekmektedir [91] [92].

Endüstri Standartlarına Uygunluk Değerlendirmesi

Sistemin mevcut tasarımını, modern backend geliştirme prensipleri ve endüstri standartları açısından incelediğimizde aşağıdaki alanlarda değerlendirmek mümkündür:

- **Clean Architecture & Katmanlı Mimari:** Proje yapısı Clean Architecture prensiplerine genel olarak uygundur. Bağımlılıklar iç katmandan dışa doğru akacak şekilde düzenlenmiştir; domain katmanı dış katmanlardan izoledir ve yalnızca arayüzler tanımlar [27]. Katmanlar net bir sorumluluk ayrımıyla (UI/Controller, Service, Domain Model, Infra) oluşturulmuştur [93] [94]. Bu, **Separation of Concerns** ve **Dependency Inversion** ilkelerinin uygulandığını gösterir. Domain-Driven Design yaklaşımı da benimsenmiştir: Domain modelleri gerçek iş kavramlarını yansıtır ve kendi davranışlarını içerir (ör. AlertRule içinde tetiklenme kontrolü, User içinde şifre hash doğrulama kuralı vb. düşünülmüş). Repository deseninin kullanımı, domain ile veri erişimi detaylarının ayrışmasını sağlamaktadır [95]. Bu mimari tercih, endüstride **bakım kolaylığı** ve **test edilebilirlik** açısından best practice olarak kabul edilir ve projede başarıyla uygulanmıştır.

- **Domain-Driven Design (DDD):** Proje, DDD'nin temel prensiplerini takip ediyor. **Zengin domain modelleri** (Rich Domain Model) yaklaşımı sayesinde, iş mantığı büyük oranda entity'lerin içinde kapsüllenmiş durumda [40] [41]. Örneğin kuralın tetiklenip tetiklenemeyeceğine domain nesnesi kendi karar veriyor (cooldown kontrolü vb.), böylece iş akışı kodu daha sade kalıyor. Ayrıca **Ubiquitous Language** (ortak dil) yansıtılmış: Sınıf ve değişken isimleri problem alanındaki terimlerle (Block, Transaction, AlertRule, MonitoredContract vs.) birebir örtüşüyor, bu da kodun anlaşılabilirliğini artırır. **Repository** katmanı, DDD'nin önerdiği şekilde domain ile veritabanı etkileşimini soyutlamış; birer interface olarak tanımlanan repository'ler sayesinde domain katmanı veri saklama detaylarından bağımsız çalışabiliyor [95]. Bu da birim testlerde veya ileride farklı bir veri kaynağına geçişte esneklik sağlayacaktır. Kısacası proje mimarisi DDD'nin *katmanlı mimari, entity ve value object ayrımı, repository deseni* gibi konseptleriyle uyumlu gözükmektedir.

- **Event-Driven Design & Asenkron İletişim:** Sistemin doğası kısmen event-driven diyebiliriz, zira harici bir **olay akışı (Stacks blokzincirindeki olaylar)** üzerinden tetikleniyor. Chainhook, blockchain üzerindeki yeni blok olayını yakalayıp sisteme ileten bir event kaynağı gibi çalışıyor. Backend de bu olayı alıp işlerken internal olarak bazı asenkron mekanizmalar kullanıyor (bildirim gönderimi @Async ile ayrılmış). Ancak, tam anlamıyla bir *event-driven microservice* mimarisi şu an uygulanmamış durumda. Örneğin **domain event** kavramı ya da **mesaj kuyrukları** (message broker) aracılığıyla servisler arası iletişim gibi şeyler yok – zaten sistem tek bir uygulama olarak tasarlanmış. Yine de iç süreçlerde potansiyel olarak event publishing kullanılabilir (ör. "TransactionProcessed" gibi bir ApplicationEvent tanımlayıp dinleyiciler ile alert eşleştirme ve bildirim gönderimi gibi işlemleri ayırmak mümkündü). Mevcut yapıda bildirimler, işlem işleme akışının sonunda doğrudan çağrılıyor. Bu senaryoda, **asenkronlik** kısmen var (bildirimler ayrı thread ile), fakat örneğin büyük bir payload geldiğinde tüm işlemler ve kurallar aynı işlem içinde senkron değerlendiriliyor. Endüstride yüksek ölçekli sistemlerde görülen Kafka gibi bir event stream veya komut/sorgu ayrışması (CQRS) bu MVP'de implemente edilmemiş – ancak analiz raporlarında ileride **event sourcing** ve **CQRS** desenlerinin değerlendirilebileceği önerilmiş [96] [97]. Sonuç olarak, sistem **reaktif** veya tam event-driven bir mimaride değil, ancak ihtiyaç duyulan yerlerde asenkron işleme destek verecek yapılar (Thread pool, Async vs.) kullanılmış. İleride ölçek artışı öngörülürse, **mesaj kuyrukları ile decoupling** veya **stream processing** (Kafka, RabbitMQ entegrasyonu) eklenmesi endüstri standardı bir ilerleme olacaktır.

- **Kimlik Doğrulama & Yetkilendirme (JWT, OAuth, API Keys):** Proje Spring Security'i dahil etmiş ancak JWT tabanlı kimlik doğrulama henüz tam entegre edilmemiş görünüyor [78]. Kayıt ve login

uçları olmasına rağmen, başarılı login sonucu JWT token üretilmesi ve sonraki isteklerde bu token'ın doğrulanması süreci tamamlanmamış (JwtTokenService sınıfı mevcut ancak SecurityFilterChain'de konfigürasyon eksik olabilir). Endüstri standardı olarak, bir web API'sinin JWT ile stateless auth yapması beklenir. Bu eksik halledilene kadar API'lar pratikte korumasız olabilir. Ayrıca **OAuth2/OpenID Connect** desteği, çok kullanıcı barındıran sistemlerde yaygın olsa da bu MVP kapsamında belki gerekmeyebilir; yine de uzun vadede Google, Facebook gibi Identity Provider'lar ile entegrasyon düşünülebilir. API key bazlı erişim (ör. harici servislerin belli endpoint'lere erişmesi için) şu an yok, fakat gerektiğinde eklenmesi mümkün (User entity'de API key alanı, veya ayrı bir ApiClient tablosu ile). **Yetkilendirme** tarafında ise en azından kullanıcı vs. admin rol ayrımı için bir `UserRole` enum tanımlı (USER, ADMIN) [76]. Fakat şu an sistemde admin ayrıcalığı gerektiren bir fonksiyon yok. İleride bir yönetici paneli gelirse Role bazlı kısıtlamalar SecurityConfig'te konfigüre edilebilir. Son olarak, **HMAC imzalama** konusu Chainhook için önem arz ediyor: Endüstride webhook alan servislerin *kaynak doğrulaması* yapması şarttır (örn. GitHub webhooks vs. bir secret ile imzalanır). Bu projede de HMAC imza kontrolü düşünülmüş, bir filter iskeleti yazılmış ancak bunun aktif hale getirilmesi ve secret yönetimi (config'de secret tanımı) yapılmalıdır [80]. Özetle, **güvenlik katmanı** endüstri standartlarına ulaşmak için biraz daha işleme ihtiyaç duyuyor: JWT auth tamamlanmalı, HMAC doğrulama aktif edilmeli, gerekirse rate-limiting ve audit logging ile API suistimalleri engellenmelidir.

- **HMAC & İletişim Güvenliği:** Chainhook entegrasyonu haricinde, sistemin diğer çıkış noktaları e-posta ve webhook bildirimleridir. E-posta gönderimi Spring Boot Starter Mail ile yapılır – burada TLS üzerinden SMTP kullanımı gibi konular muhtemelen Spring Boot tarafından yapılandırılıyor. Webhook gönderimleri ise sistemin başka sunuculara HTTP POST yapması demek; bu çağrıların da HTTPS üzerinden ve sertifika doğrulamalarıyla güvenli olması gerekir (muhtemelen `RestTemplate` veya WebClient ile gönderiliyor, default olarak SSL kontrolü yapılır). Endüstride webhook gönderirken de bir imzalama veya authentication mekanizması kullanılabilir (ör. hedef URL'e bir secret parametre eklemek veya payload'ı imzalamak), ancak genellikle bu hedef sistemle anlaşmaya bağlı. Bu proje kapsamında böyle bir özellik muhtemelen gerekmiyor, fakat dokümanlarda *dead-letter queue* önerisi var (eğer webhook hedefi başarısız olursa mesaj kaybolmasın) [98]. Bu, güvenlikten ziyade dayanıklılık (reliability) ile ilgili bir konu.

- **Performans ve Ölçeklenebilirlik:** Sistemin mimarisi tek bir uygulama olarak (monolitik) planlanmış görünüyor. Bu, MVP aşaması için gayet makul ve basit bir yaklaşım. Performans kritik noktaları olarak; veritabanı dizin stratejisi, kural eşleştirme algoritması ve batch işlem yönetimi göze çarpıyor. Endüstri standardı olarak, yoğun insert yapılan tablolarda doğru indeksleme ve gerekirse **partisyonlama** önemlidir. Bu projede TransactionEvent tablosunda 8 indeks, ContractCall'da 5 indeks tanımlı olması, okuma performansı için düşünülmüş [89] ancak yazma performansına etkisi olabileceğinden dikkat gerektirir (Her yeni event insert'inde bu indeksler güncellenecek, çok yüksek hacimde insert varsa gecikme yaratabilir). Çözüm olarak ileride **partial indexes** veya **table partitioning** uygulanması öneriliyor ki bu endüstride yaygın bir yaklaşımdır (büyük tabloları tarih bazında bölmek, eski veriyi arşivlemek vs.) [99] [100]. Kural eşleştirme algoritması O(n) olarak kalmış, ki bu n (aktif kural sayısı) büyüdükçe ölçeklenemez hale gelir – endüstride bu tip pattern matching problemleri için Rete algoritması kullanan kural motorları (Drools, Easy Rules) veya en azından etkin bir ön filtreleme (indexing, Bloom filter) yapılır [101] [102]. Burada önerilen bellek içi indeksleme (kuralları ilgili kontrat veya event tipine göre gruplayarak eşleştirmek) bir nevi ön filtreleme olup performansı iyileştirecektir. Ayrıca sistemin **asenkronlaşmaya** uygun noktaları var: Chainhook çağrısı geldiğinde anında 200 OK dönüp işlemleri arkaya almak (queue) daha ölçeklenebilir olur. Şu an synchronous olarak veritabanına yazıp tüm kuralları değerlendirip belki de bildirim gönderip öyle cevap dönüyor olabilir. Endüstride bu tip gerçek zamanlı sistemlerde, **yüksek throughput** için genellikle gelen

olaylar kuyruklanır (Kafka gibi) ve arka planda işlenir, API çağrısına hemen cevap verilir. Bu MVP'de henüz o seviyede bir ayrıştırma yok, ancak Spring @Async ile benzer bir işlem yapılıp Chainhook endpoint'i çağrıyı bekletmeden kapatabilir (dokümanda "Chainhook endpoint hemen 200 dönmeli, processing queue'ya at" şeklinde öneri var) [103] [104]. Yine ölçeklenebilirlik adına, **veritabanı bağlantı havuzu** ayarları önemlidir; HikariCP default olarak 10 bağlantı ile gelir, burada belki 20 olarak ayarlanmış (application.yml içinde görmedik ama dokümanda örnek parametreler verilmiş) [105] [106]. Sonuç olarak, mimari tek bir instance üzerinde çalışacak şekilde ancak yatay genişlemeye de uygun (state yok denecek kadar az, JWT stateless auth olursa). Yatay skalada cache'lerin (özellikle AlertRule cache) **dağıtık** olması gerekebilir (Redis gibi bir external cache ile). Şu anki çözüm `ConcurrentHashMap` ile lokal bellek cache'ine benziyor, bu tek instance için ok fakat birden fazla instance olursa tutarsızlık olabilir. Endüstri standardı, çoklu instance için paylaşımlı bir cache kullanmaktır (Redis, Hazelcast vb).

- **Hata Toleransı ve Gözlemlenebilirlik:** Production ortamında bir sistemin dayanıklılığı ve izlenebilirliği çok kritik. Bu projede bazı dayanıklılık mekanizmaları eksik: Örneğin **retry/circuit breaker** uygulanmaması, başarısız üçüncü parti çağrıların (ör. webhook gönderimi) sistemi zor durumda bırakmasına yol açabilir. Endüstride bu tip entegrasyon noktalarında *Resilience4j* veya benzeri kütüphaneler kullanılarak otomatik yeniden deneme, exponential backoff, circuit breaker (çok sayıda hata olursa bir süre denememe) gibi desenler uygulanır. Dokümanda WebhookNotificationService için örnek bir @Retryable ve @CircuitBreaker anotasyonlu çözüm önerilmiş [23] [107] – bunlar Spring Retry ve Resilience4j ile entegre çalışan yapılar, bunların kullanımı endüstride best practice kabul edilir. Gözlemlenebilirlik tarafında ise Actuator endpoints (sağlık, metrikler) açmak temel bir gereksinimdir. Prometheus ile metrik toplama hazır ancak özel uygulama metrikleri eklemek de önemli (ör. **Counter**: kaç bildirim gönderildi, **Timer**: bir işlem ortalama ne kadar sürüyor vs.). Kodda Micrometer entegre edilmiş olsa da henüz bu metriklerin tanımlanmadığı görülüyor [84]. Endüstride, en azından kritik süreçler için bu metrikler kod içine eklenir ve dashboard'lar kurulur. Dağıtık izleme (Tracing) ise bir monolitik uygulamada opsiyonel olabilir, ancak sistem büyür veya microservice mimarisine evrilirse isteklerin izini sürmek için gerekli olacaktır. Spring Boot 3 ile Micrometer Tracing/Sleuth gibi araçlar entegre edilebilir. Logging konusunda JSON formatlı loglama, korelasyon ID'leri, hata durumlarında stack trace yakalama gibi konulara dikkat edilmeli; özellikle API giriş/çıkışlarının audit log'lanması (kimin ne zaman hangi kuralları eklediği, vs.) güvenlik açısından faydalıdır.

- **Test Edilebilirlik:** Projede test bağımlılıklarının eklenmiş olması (Spring Boot Starter Test, JUnit, Testcontainers PostgreSQL vs.) sevindirici [60]. Bu, geliştiricinin test yazmayı planladığını gösteriyor. Domain katmanı zengin olduğu için **birim testleri** (unit test) yazmak kolay olmalı; örneğin AlertRule.canTrigger() veya AlertMatchingService gibi sınıflar için saf Java ile testler yazılabilir. **Entegrasyon testleri** için ise Testcontainers ile gerçek PostgreSQL üzerinde uçtan uca senaryolar koşmak mümkün (zaten dependency eklenmiş). Endüstri standardı olarak, özellikle bu gibi kritik sistemlerde her bir kural türü ve bildirim akışı için testler olmalı. Örneğin "büyük bir token transfer eventi geldiğinde ilgili kural tetikleniyor mu, email servisinin send metodu çağrılıyor mu" gibi uçtan uca testler kurgulanabilir. Şu an testlerin kapsamı belirsiz; belki henüz yazılmamış olabilir. Production-ready için testlerin kapsamını artırmak, CI/CD sürecine testleri entegre etmek (her build'de çalışması) çok önemlidir. Ayrıca performans testleri ve yük testleri de ileride düşünülmeli; özellikle kural eşleştirme ve veri tabanı yazma performansı, beklenen en yüksek yükte test edilmelidir.

Teknik Önceliklerin Belirlenmesi (Performans, Güvenlik, Ölçeklenebilirlik, Bakım)

Mevcut sistem mimarisi göz önüne alındığında, **production** ortamında başarılı olabilmesi için aşağıdaki teknik öncelikler öne çıkmaktadır:

1. **Güvenlik (Security) – Öncelik: Çok Yüksek (Kritik).** Şu anda API güvenlik katmanı tam anlamıyla devrede değil; bu durum hem sistemi dış erişimlere karşı savunmasız bırakır hem de yanlış kullanımlara açıktır. Bu nedenle **ilk öncelik**, JWT tabanlı kimlik doğrulamanın eksiksiz uygulanması, kullanıcı parolalarının güvenli şekilde (BCrypt) saklanması, rol bazlı erişim kontrolünün eklenmesi ve Chainhook webhook'larının HMAC ile doğrulanmasının sağlanması olmalıdır [108] [109]. Ayrıca, API suistimallerini önlemek için rate limiting (IP bazlı istek sınırları) ve audit logging (kritik işlemlerin kaydı) gibi güvenlik önlemleri de erkenden ele alınmalı. Güvenlik alanında bir açık kalması, tüm sistemin güvenilirliğini sorgulatacağı için birinci derecede önceliklidir.

2. **Ölçeklenebilirlik & Performans – Öncelik: Yüksek.** Blockchain izleme sistemleri potansiyel olarak yüksek hacimli veri akışıyla karşılaşır. Bu projede performansı tehdit eden başlıca konu, **alert kural eşleştirme** ve **veri ekleme hızı** olarak görünüyor. Aktif kural sayısı arttığında her bir yeni işlem için kuralların teker teker kontrol edilmesi mevcut haliyle ölçeklenebilir değil (O(n) algoritma) [110]. Ayrıca günde on binlerce event eklenirse veritabanı yazma işlemleri (özellikle indeks güncellemeleri) bir darboğaz oluşturabilir [111] [112]. Dolayısıyla ikinci öncelik, sistemin **yük altında verimli çalışmasını sağlamak** olmalı. Bunun için kural eşleştirme mekanizmasının optimize edilmesi (önbellekleme, indeksleme veya kural motoru entegrasyonu), yoğun insert'ler için veritabanı tarafında partisyonlama/indeks optimizasyonu, büyük JSON payload'ların işlenmesinde bellek ve zaman optimizasyonları (gerekirse streaming parse, chunked processing) gerekecektir. Bu iyileştirmeler yapılmazsa sistem artan yükle birlikte yavaşlayacak veya veri kayıplarına yol açabilecektir.

3. **Dayanıklılık ve Tutarlılık (Reliability & Data Integrity) – Öncelik: Yüksek.** Sistem her ne kadar blockchain'den besleniyor olsa da kendi içinde tutarlılığı sağlamak zorunda. Örneğin bir blockchain **reorg** durumunda (geriye dönük blok iptali) sistem veritabanında o blokla ilgili kayıtları geri alabilmelidir – şu an bu senaryo için yalnızca bir "TODO" notu mevcut, henüz uygulanmamış [113]. Benzer şekilde, bildirim gönderiminde bir hata oluştuğunda bu bildirim kaybedilmemeli, sistem hatalara dayanıklı olmalıdır (retry mekanizması, dead-letter queue eksikliği şu an bir risk) [22] [98]. Transaction işlemlerinin yarısında hata olsa dahi sistem tutarlı kalmalı (ya hep ya hiç). Şu anki implementasyonda transactional boundary'lerin dikkatli ayarlanması gerekiyor; örneğin bir blok ve işlemleri kaydedip event kayıtları başarısız olursa, veritabanında yarım bir işlem seti kalmamalı [114] [115]. Bütün bu konular sistemin güvenilirliği açısından kritik. Bu nedenle üçüncü öncelik, **hata toleransını artırmak** ve **veri tutarlılığını sağlayacak önlemleri** hayata geçirmek olmalı. Bu, kod seviyesinde ekstra kontroller, try-catch'ler, transaction yönetimi, geri dönüş senaryoları (rollback) gibi konuları içerir.

4. **Gözlemlenebilirlik (Observability) ve Operasyonel İzleme – Öncelik: Orta.** Production ortamında bir sorunu çözebilmek için önce onu tespit edebilmek gerekir. Bu nedenle sistemin **izlenmesi** (metrics, logging, tracing) büyük önem taşır. Mevcut durumda temel altyapı var ancak özel metrikler yok, loglar muhtemelen yeterince detaylı değil. Bu, güvenlik ve performans kadar *acil* olmasa da ihmal edilmemesi gereken bir alandır. Dördüncü öncelik olarak, **monitoring & observability** ele alınmalı: Uygulamaya özel metrikler eklenmeli (ör. saniyede işlenen event sayısı, kural eşleştirme süresi, bildirim başarı/oranları vs. gibi), Prometheus/Grafana gibi

araçlarla takip edilmeli [116] [117]. Ayrıca distributed tracing kurgusu ileride gerekebilir (özellikle mikroservis mimarisine geçiş olursa) [118]. Logların analizini kolaylaştırmak için JSON format tercih etmek, log seviyelerini doğru ayarlamak (INFO, DEBUG, ERROR ayrımları) ve kritik aksiyonları (login, kural oluşturma, vs.) audit log'a yazmak da gereklidir. Bu öncelik, ilk üçünün ardından devreye alınsa da planlama aşamasında düşünülmelidir; çünkü bir sorun olduğunda geç kalmadan teşhis edebilmek için bu altyapı şarttır.

5. **Bakım Kolaylığı ve Genişleyebilirlik (Maintainability & Extensibility)** – **Öncelik: Orta.** Sistem halihazırda temiz bir mimariye sahip olduğu için bakım ve genişletme konusunda avantajlı. Ancak ileride ekip büyür, kod tabanı genişlerse, temiz kod prensiplerine uyumun sürdürülmesi gerekir. Bu kapsamda kod kalitesini güvence altına almak için **kod gözden geçirme** pratikleri, **otomatik testler** ve belki static code analysis araçları (SonarQube gibi) devreye alınmalı. Dokümantasyonun güncel tutulması (özellikle API dökümü ve önemli mimari kararların kaydı) önemlidir. Sistemin yeni özelliklere (örn. yeni alert türleri, yeni bildirim kanalları) açık olması hedeflenmiştir ve bu mimari bunu destekliyor [68]. Bunu korumak adına, eklenecek her yeni özelliğin mevcut katmanlaşmaya uygun tasarlanması (Doğru pakete/doğru katmana ekleme) ve bağımlılıkların kontrol edilmesi gerekir. Bakım tarafında bir diğer konu da **versiyonlama** ve **deployment** kolaylığıdır: Prod ortamda kesinti yaşamadan yeni sürüm çıkarabilmek için containerization (Docker) ve otomasyon (CI/CD) önemli rol oynar. Bu alanlar da genişleyebilirlik başlığı altında değerlendirilmeli.

6. **Uyum ve Diğer Öncelikler:** Yukarıdakilere ek olarak, **uyumluluk (compliance)** gereksinimleri (örn. GDPR açısından loglarda hassas veri tutmama, vs.), **yüksek erişilebilirlik** (multi-node deployment, failover) gibi konular proje olgunlaştıkça gündeme gelecektir. MVP aşamasında bunlar ikincil olsa da, mimari kararlar alınırken (ör. stateless tasarım, external cache, vs.) gelecekteki bu ihtiyaçlar göz önüne alınmıştır diyebiliriz.

Öncelikleri özetlemek gerekirse: Önce **güvenlik temelleri** atılmalı, ardından **performans ve ölçeklenebilirlik engelleri** kaldırılmalı, takiben **sistem güvenilirliği ve gözlemlenebilirliği** güçlendirilmeli. Bu adımlar atıldıktan sonra sistem gerçek dünya yüküne ve saldırılara karşı dayanıklı hale gelecek, geliştirici ekip de sistemi rahatlıkla izleyip genişletebilecektir.

Eksikler, Riskler ve İyileştirme Önerileri

Mevcut yapının detaylı analizinde tespit edilen eksik veya riskli noktalar aşağıda sınıflandırılmış ve her biri için **çözüm önerileri** sunulmuştur. Öneriler, mimari ve kod seviyesinde iyileştirmeleri içerir ve mümkün olduğunca **örnek kod** parçaları ile desteklenmiştir.

Kritik Düzeyde Eksikler / Riskler

1. JWT Tabanlı Kimlik Doğrulama Eksikliği (Security) – *Risk:* **Çok yüksek.** Şu anda sistemde kullanıcının login olabileceği API mevcut ancak login sonucunda JWT token üretimi ve sonraki isteklerde bu token'ın doğrulanması tam uygulanmamış. Bu da API'ların pratikte korumasız olması anlamına geliyor. Örneğin `/api/v1/alerts/rules` gibi uçlar herkes tarafından çağrılabilir durumda olabilir. Ayrıca parolaların güvenliği de kritik; `User.passwordHash` alanı olsa da bunun gerçekten bcrypt hash tutup tutmadığı kontrol edilmeli. *Çözüm:* Spring Security konfigürasyonu tam olarak kurulmalıdır. **SecurityConfig** içinde JWT filtreleri eklenecek, tüm korumalı API'lar authentication gerektirecek şekilde ayarlanmalıdır. Örnek olarak:

```
@EnableWebSecurity
@Configuration
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http
          .csrf().disable()
          .authorizeHttpRequests(auth -> auth
              .requestMatchers("/api/v1/webhook/chainhook").permitAll()
              .requestMatchers("/api/v1/auth/**").permitAll()
              .anyRequest().authenticated()
          )
          .addFilterBefore(jwtAuthenticationFilter(),
UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
    @Bean
    public JwtAuthenticationFilter jwtAuthenticationFilter() {
        return new JwtAuthenticationFilter(jwtTokenProvider);
    }
}
```

Yukarıdaki gibi bir SecurityConfig ile, login/register ve chainhook endpoint'leri dışındaki tüm istekler JWT doğrulamasına tabi olur. JWT üreten ve doğrulayan **JwtTokenProvider** (veya JwtTokenService) sınıfı gerçeklenmelidir. Bu sınıf HS256 algoritması ile imzalama, expire time, vs. içerebilir. Ayrıca user login olduğunda AuthenticationController içinde:

```
@PostMapping("/api/v1/auth/login")
public ResponseEntity<AuthenticationResponse> login(@RequestBody
LoginRequest req) {
    User user = authService.login(req.getEmail(), req.getPassword());
    String token = jwtTokenProvider.generateToken(user);
    return ResponseEntity.ok(new AuthenticationResponse(token));
}
```

şeklinde token oluşturup döndürmek gerekir. Bu şekilde istemci her istekte Authorization header'da "Bearer <token>" gönderebilir, ve JwtAuthenticationFilter bunu kontrol ederek Spring Security context'ine kullanıcıyı set eder.

Ayrıca, **password hashing** mutlaka yapılmalıdır. Kayıt sırasında `authService.register()` içinde plaintext parolayı `BCryptPasswordEncoder` ile hash'leyip User.passwordHash olarak saklamak gerekir. Login olurken de girilen parolayı aynı encoder ile karşılaştırmak şarttır. Örneğin:

```
@Service
public class AuthenticationService {
    @Autowired private UserRepository userRepo;
    @Autowired private PasswordEncoder passwordEncoder;
```

```
    public User register(String email, String rawPassword, ...) {
        User user = new User();
        user.setEmail(email);
        user.setPasswordHash(passwordEncoder.encode(rawPassword));
        // ... diğer alanlar
        return userRepo.save(user);
    }
    public User login(String email, String rawPassword) {
        User user = userRepo.findByEmail(email)
            .orElseThrow(() -> new BadCredentialsException("Bad creds"));
        if (!passwordEncoder.matches(rawPassword, user.getPasswordHash())) {
            throw new BadCredentialsException("Bad creds");
        }
        return user;
    }
}
```

Bu şekilde basit ama güvenli bir auth akışı sağlanabilir. Spring Security için PasswordEncoder bean'i olarak BCrypt konfigüre etmek yeterlidir (Strength 10+ önerilir).

Ek öneriler: - **Refresh token** mekanizması (uzun süreli oturumlar için) ileride gerekebilir. - **Rol tabanlı yetki**: Şu an sadece USER rolü var. Admin yetkileri olacaksa, SecurityConfig'te `.hasRole("ADMIN")` gibi kısıtlar eklenmeli ve User nesnesine role alanı eklenmeli (ki enum'da zaten ADMIN tanımlı [76]). - **OAuth2**: MVP için ikincil öncelik ama sosyal login veya kurumsal SSO gibi ihtiyaçlar olursa Spring Security OAuth modülü entegre edilebilir.

2. Webhook Güvenliği ve Rate Limiting Eksikliği – *Risk:* **Yüksek.** Chainhook'tan gelen verinin güvenliği kısmen ele alınmış ama tam uygulanmamış. Şu an `ChainhookHmacFilter` var ancak SecurityConfig içinde aktif mi belirsiz. Ayrıca sistemin public endpoint'lerine karşı herhangi bir istek limitleme yok. Bu, kötü niyetli birinin ardışık API çağrılarıyla sistemi zorlamasına yol açabilir. *Çözüm:*

- **HMAC İmza Doğrulama:** Chainhook tarafında verilen bir secret (paylaşılan anahtar) olmalıdır. Bu secret, backend konfigürasyonunda tutulur (örn. `chainhook.secret`). WebhookController çağrıldığında HTTP header'ında gelen imza (mesela `X-Chainhook-Signature`) ve request body verisi birlikte kullanılarak HMAC-SHA256 hesaplanır ve eşleştirilir [80]. Bunu yapacak bir bileşen (ChainhookSignatureValidator) zaten önerilmiş [80], onu kullanarak filtre içerisinde kontrol yapmalıyız. Eğer imza geçerli değilse istek reddedilmeli (403 Forbidden). Bu sayede sadece Chainhook'ın gönderdiği gerçek blockchain event'leri işlenecek, başka kaynaklardan gelebilecek sahte istekler ayıklanacaktır. Aşağıda konsept kod görülüyor:

```
@Component
public class ChainhookHmacFilter extends OncePerRequestFilter {
    @Value("${chainhook.secret}")
    private String chainhookSecret;
    @Autowired
    private ChainhookSignatureValidator validator;
    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain chain) throws ServletException,
IOException {
```

```
        if (request.getRequestURI().contains("/api/v1/webhook/chainhook")) {
            String signature = request.getHeader("X-Chainhook-Signature");
            String payload =
request.getReader().lines().collect(Collectors.joining());
            if (!validator.validateSignature(payload, signature,
chainhookSecret)) {
                response.sendError(HttpStatus.FORBIDDEN.value(), "Invalid
signature");
                return;
            }
        }
        chain.doFilter(request, response);
    }
}
```

Yukarıdaki filtre, chainhook endpoint'ine gelen isteğin imzasını doğrular. Bunu SecurityConfig'te `.addFilterBefore(chainhookHmacFilter, JwtAuthenticationFilter.class)` diyerek zincire eklemek gerekir, ki JWT'den bile önce çalışsın (zaten chainhook çağrısı public olacağı için JWT aranmayacak ama imza aranacak).

- **Rate Limiting:** API abuse'ını engellemek için IP bazlı veya kullanıcı bazlı hız limitleri koymak önemli. Spring için doğrudan bir rate-limit yok ama Bucket4j veya Resilience4j RateLimiter kullanılabilir. Örneğin, **Bucket4j** ile bir servlet filter şeklinde basit bir token bucket uygulanabilir. Her IP için saniyede X request limiti konulabilir. Kod örneği:

```
@Component
public class RateLimitFilter extends OncePerRequestFilter {
    private final Map<String, Bucket> buckets = new ConcurrentHashMap<>();
    private final Refill refill = Refill.greedy(10,
Duration.ofMinutes(1)); // dakikada 10 istek
    private final BucketConfiguration config =
BucketConfiguration.builder().addLimit(Bandwidth.classic(10,
refill)).build();
    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {
        String ip = request.getRemoteAddr();
        Bucket bucket = buckets.computeIfAbsent(ip, k ->
Bucket4j.newBucket(config));
        if (bucket.tryConsume(1)) {
            filterChain.doFilter(request, response);
        } else {
            response.sendError(HttpStatus.TOO_MANY_REQUESTS.value(), "Rate
limit exceeded");
        }
    }
}
```

Bu örnek dakikada 10 istek izni veriyor IP başına. İhtiyaca göre oranlar ayarlanabilir. Bu filtre de SecurityConfig'e eklenmelidir (genelde en başa). Alternatif olarak, CloudFlare gibi dış servisler veya API Gateway kullanılarak da rate-limit yapılabilir. **Unutulmamalı ki**, rate-limit uygularsak Chainhook gibi güvenilir kaynakların IP'lerini muaf tutmak gerekebilir.

3. Alert Kural Eşleştirme Performansı (O(n) Algoritma) – *Risk:* **Çok yüksek (Scalability Killer).** Sistemdeki belki de en büyük performans riski budur. Mevcut durumda her yeni transaction işlendiğinde, tüm aktif alert rule listesini döngüyle gezip `matchesRule(transaction, rule)` kontrolü yapılıyor [14]. Bu, kural sayısı n ve gelen işlem sayısı m ise, en kötü O(m*n) karmaşıklık demek. Örneğin 1000 kural ve günde 20,000 işlem olsa 20 milyon kural kontrolü anlamına gelir ki bu sürdürülebilir değil [110] [119]. Çözüm:* Kural eşleştirme motorunun optimizasyonu şart. Birkaç yaklaşım birleştirilebilir:

- **Kural Önbelleği ve İndeksleme:** AlertRule'ları değerlendirirken her seferinde veritabanından çekmek yerine zaten bellek içinde tutuyoruz (invalidateRulesCache() metodu var, demek ki bir cache mekanizması düşünülmüş) [120]. Bunu daha da geliştirelim: Kuralları *anahtar alanlara göre gruplayarak* bir Map yapısında tutabiliriz. Örneğin contractIdentifier veya eventType, hatta her ikisi. Bir `AlertRuleCache` servisi, periyodik olarak (örn. her 1 dakikada bir) aktif kuralları DB'den çekip aşağıdaki gibi yapılar kurabilir:

```
@Service
public class AlertRuleCache {
    private Map<String, List<AlertRule>> rulesByContract = new
ConcurrentHashMap<>();
    private Map<EventType, List<AlertRule>> rulesByEventType = new
ConcurrentHashMap<>();
    @Autowired AlertRuleRepository repo;
    @Scheduled(fixedRate = 60000)
    public void refreshCache() {
        List<AlertRule> activeRules = repo.findAllActive();
        rulesByContract.clear();
        rulesByEventType.clear();
        for (AlertRule rule : activeRules) {
            String contract = rule.getMonitoredContract() != null ?
rule.getMonitoredContract().getContractIdentifier() : "*";
            rulesByContract.computeIfAbsent(contract, k -> new
ArrayList<>()).add(rule);

rulesByEventType.computeIfAbsent(rule.getRuleType().getEventType(), k -> new
ArrayList<>()).add(rule);
        }
    }
    public List<AlertRule> getRulesForEvent(EventType eventType, String
contractId) {
        List<AlertRule> result = new ArrayList<>();
        result.addAll(rulesByContract.getOrDefault(contractId,
Collections.emptyList()));
        result.addAll(rulesByContract.getOrDefault("*",
Collections.emptyList())); // wildcard rules
        // belki hem contract hem event tipini birlikte değerlendirmek lazım
        result.retainAll(rulesByEventType.getOrDefault(eventType,
Collections.emptyList()));
```

```
        return result;
    }
}
```

Yukarıdaki pseudo-kod iki türlü indeksleme yapıyor: kontrat ID'ye göre ve event tipine göre. Sonra getRulesForEvent ile ilgili kontratın (veya genel geçer * kural varsa) ve event tipinin kesişimini alıyor. Bu sayede eğer bir event belirli bir kontratta olduysa, sadece o kontrata özel kurallar ve genel kurallar dönecek, diğer yüzlerce kural elenecek. Bu yöntem özellikle kurallar spesifik kontratlara odaklıysa muazzam iyileştirme sağlar. Eğer kurallar daha karmaşık koşullar içeriyorsa (örn. miktar threshold, sender adresi vs.) onlarda hala if kontrol yapacağız ama en azından çoğu alakasız kuralı elemiş olacağız.

- **Polimorfik Kural Yapısı:** Bir diğer iyileştirme, AlertRule entity'sini alt tiplere ayırmak. Şu an JSON string içinde koşullar saklanıyor (conditions alanı) ve runtime parse ediliyor. Bunu daha tip-güvenli hale getirirsek, her kural tipini ayrı sınıf yapıp spesifik alanlar tanımlayabiliriz (zaten kural tiplerimiz belli: ContractCallAlertRule, TokenTransferAlertRule vb.). Bu alt sınıfların içinde `matches(event)` gibi metodlar olabilir. Bu sayede koşul kontrolü daha direkt ve hızlı yapılır, JSON parse yükü olmaz. Örneğin:

```
@Entity @DiscriminatorValue("CONTRACT_CALL")
public class ContractCallAlertRule extends AlertRule {
    private String contractIdentifier;
    private String functionName;
    private BigDecimal amountThreshold;
    @Override
    public boolean matches(TransactionEvent event) {
        if (!(event instanceof ContractCall)) return false;
        ContractCall call = (ContractCall) event;
        return call.getContractIdentifier().equals(contractIdentifier)
            && call.getFunctionName().equals(functionName)
            && call.getFunctionArgsRaw() != null // args dolu mu gibi
kontroller
            && call.getAmount() != null
            && call.getAmount().compareTo(amountThreshold) >= 0;
    }
}
```

Bu örnekle, JSON parse etmeden doğrudan alan üzerinden kontrol yapılabilir. (Not: Mevcut modelde ContractCall detayları Transaction içindeydi, belki TransactionEvent alt tipine de ayrılabilir). Polimorfik yapı bellek kullanımını biraz artırsa da okunabilirlik ve tip güvenliği sağlar. Ayrıca repository sorgularında `ruleType` ile filtreleme kolaylaşır (SINGLE_TABLE stratejisinde diskriminator ile).

- **Kural Motoru Entegrasyonu:** Uzun vadede kural sayısı ve karmaşıklığı artarsa, **Drools** gibi bir kural motoru entegre etmek düşünülebilir. Drools, Rete algoritmasıyla benzer kural şartlarını paylaşan yapıları optimize eder. Fakat MVP için belki gereksiz karmaşa olabilir. Daha hafif bir opsiyon olarak **Easy Rules** gibi kütüphaneler de var. Şu an bellek içi indeksleme yeterli gelebilir. Yine de endüstri standardı bir çözüm not etmek gerekirse: Rete-based engine kullanmak ve kural tanımlarını ya Java kodu yerine kural DSL'inde tutmak (kullanıcı arayüzü de belki ileride) sistemi daha esnek kılar.

- **Bloom Filter ile Ön Eleme:** Dokümandaki bir başka öneri Bloom filter kullanımıydı [102]. Örneğin sık gelen event özellikleri (kontrat adresi vs.) için Bloom filter ile "hiç kural var mı yok mu" hızlı kontrol edilebilir. False positive üretme pahasına, eğer kesinlikle kural yok diyebiliyorsa direkt eleyebilir. Bu daha ileri seviye bir optimizasyon, belki şu aşamada şart değil ama bahse değer.

Önerilen önbellek ve indeksleme yaklaşımı ile, **performans kazancı** dramatik olacaktır. Bu aynı zamanda sistemin ölçeklenebilirliğini artırır. Unutulmamalı, cache'in güncellenmesi meselesi var: Kural listesi değiştiğinde (yeni kural eklenmesi veya var olanın devre dışı bırakılması) cache invalidasyonu yapılmalı. Zaten AlertRuleService'de `invalidateRulesCache()` gibi bir metod listelenmiş [120], bu da bu ihtiyacın düşünüldüğünü gösterir. Bunu tetiklemeyi unutmayalım (örn. her createRule, updateRuleStatus çağrısında cache temizle veya refresh).

4. Bildirim Gönderiminde Retry ve Hata Yönetiminin Olmaması – ***Risk:*** Yüksek. **Sistemde uyarı bildirimleri asenkron gönderiliyor ancak gönderim başarısız olursa ne olacağı tam net değil. Mevcut implementasyon, @Async ile ayrı bir thread'de `webhookService.sendAlert(url, notification)` çağırıyor ve başarısızlık durumunda log basıp geçiyor gibi [22]. Bu durumda network kesintisi, e-posta servisinin geçici hatası gibi durumlarda bildirim iletilemeyebilir ve sonsuza dek kaybolabilir. Bu hem güvenilirlik sorunudur hem de kullanıcılar kritik bir uyarıyı alamayabilir. *Çözüm:*** Otomatik yeniden deneme (retry) **ve** circuit breaker** mekanizmalarını entegre etmek gerekiyor. Spring Retry kütüphanesi ve Resilience4j bu konuda yardımcı olabilir.

Dokümanda sunulan çözüm gayet açık [23] [107]: WebhookNotificationService metoduna @Retryable eklenip belirli Exception'lar için (ör. `RestClientException`) 3 deneme 2^n backoff'la yapılabilir. Ayrıca Resilience4j @CircuitBreaker ile eğer arka arkaya çağrılar hata veriyorsa bir süre devre dışı bırakılabilir ve *fallbackMethod* tanımlanabilir. Yukarıda kod alıntısını görmüştük, burada bir özet verelim:

```
@Service
public class WebhookNotificationService {
    @Autowired RestTemplate restTemplate;
    @Retryable(value = { RestClientException.class }, maxAttempts = 3,
               backoff = @Backoff(delay = 1000, multiplier = 2))
    @CircuitBreaker(name = "webhookService", fallbackMethod =
"sendToDeadLetter")
    public void sendAlert(String url, AlertNotification notification) {
        restTemplate.postForEntity(url, buildPayload(notification),
Void.class);
    }
    private void sendToDeadLetter(String url, AlertNotification
notification, Exception e) {
        log.error("Failed to send webhook after retries, sending to DLQ", e);
        deadLetterQueueService.enqueue(notification);
    }
}
```

Yukarıdaki kod, bir webhook göndermeyi 3 kez deneyecek (1s bekle, sonra 2s, sonra 4s gibi) eğer hep hata alırsa circuit breaker devreye girecek ve fallback olarak `sendToDeadLetter` çağrılacaktır. Bu fallback'de biz hatalı bildirimi bir *Dead Letter Queue*'ya koyuyoruz. Bu DLQ, kalıcı olarak disk/veritabanında tutulabilir veya bir mesaj kuyruğu (RabbitMQ, ActiveMQ) da olabilir. Basit yaklaşım: `dead_letters` diye bir tabloya kaydedip daha sonra bir manuel müdahale veya ayrı bir işlem ile

tekrar denemek. Endüstride daha profesyonel yaklaşım, RabbitMQ gibi bir kuyruğa atıp orada TTL sonrası yeniden asıl kuyruğa koymak veya ayrı bir consumer'ın bu DLQ'yu dinleyip alarm vermesidir.

Aynı mantık **EmailNotificationService** için de geçerli. Orada belki RestTemplate değil JavaMailSender kullanılıyor, ama benzer şekilde try-catch ve Retry uygulanabilir.

Bunların devreye alınmasıyla sistem geçici sorunlarda dayanaklı olacak, kalıcı sorunlarda ise veriyi kaybetmek yerine güvenli bir yere park edecektir. Bu, özellikle finansal önemi olabilecek uyarılar için kritiktir.

5. Blockchain Reorg (Rollback) Durumunun Ele Alınmaması – ***Risk:*** Yüksek. **Stacks blockchain (Bitcoin gibi) reorg yaşayabilir yani bir blok zincirden çıkarılıp yerine başka blok gelebilir. Chainhook, böyle bir durumda *rollback events* listesiyle eski blokları bildirebiliyor [113]. Mevcut sistemde bu durumda sadece loglama yapılıyor ve "TODO: implement rollback" notu var [121]. Eğer bu uygulanmazsa, sistem veritabanında artık geçersiz olan blokları ve işlemleri tutmaya devam eder, hatta belki yanlış alarmlar üretmiş olur. *Çözüm:*** RollbackHandler** adında bir bileşen öneriliyor [122]. Bu handler, Chainhook'un gönderdiği rollback listesinde gelen her blok için:

- İlgili **StacksBlock** kaydını bulup işaretlemeli (örneğin bir `isValid` veya `deletedAt` alanı kullanılarak) ya da tamamen silmeli [123] [124].
- O bloğa bağlı tüm **StacksTransaction** kayıtlarını da benzer şekilde geçersiz kılmalı veya silmeli [125].
- Bu işlemlerden tetiklenmiş tüm **AlertNotification** kayıtlarını da işaretlemeli (örn. `isInvalidated=true` gibi bir alan ekleyip) [126].
- Gerekirse kullanıcıya "önceki gönderilen şu uyarılar geçersiz oldu" şeklinde bir bildirim göndermeli (opsiyonel, ama tam bir proaktif sistem için bu düşünülebilir) [127].

Transactional olarak bakarsak, bu işlemlerin hepsi bir bütün olarak yapılmalı. Yani rollback event işlenirken, ilgili block, transactions, notifications güncellenip commit edilmeli. Bu belki tek tek yapılabilir ama daha pratik bir yol, eğer sadece soft-delete yapıyorsak, bir flag koymak yeterli. Silmek istersek, foreign key kısıtları vs. yüzünden sıralı silmek lazım (önce notifications, sonra transactions, sonra block). İlişkiler *cascade* ile kurulmamış olabilir çünkü normal akışta silme yok, o yüzden elle silmek gerekebilir.

Önerilen kod kabaca şöyle idi:

```
@Service
public class RollbackHandler {
    @Autowired StacksBlockRepository blockRepo;
    @Autowired StacksTransactionRepository txRepo;
    @Autowired AlertNotificationRepository alertRepo;
    @Transactional
    public void handleRollback(List<RollbackDto> rollbackEvents) {
        for (RollbackDto rb : rollbackEvents) {
            long height = rb.getBlockIdentifier().getIndex();
            StacksBlock block = blockRepo.findByBlockHeight(height)
                 .orElseThrow(() -> new IllegalStateException("Block not
found"));
            block.setIsValid(false);
            blockRepo.save(block);
            List<StacksTransaction> txs =
```

```
txRepo.findByBlockId(block.getId());
            txs.forEach(tx -> tx.setIsValid(false));
            txRepo.saveAll(txs);
            List<AlertNotification> alerts =
alertRepo.findByBlockId(block.getId());
            alerts.forEach(alert -> alert.setIsInvalidated(true));
            alertRepo.saveAll(alerts);
            // (Optional) Send notifications about invalidation
        }
    }
}
```

Bu pseudo-code'da block, transaction ve notification tablolarında boolean sahalar olduğu varsayılıyor. Şayet bu sahalar yoksa, `deletedAt` zaman damgasını set ederek de yapabiliriz (zaten çoğu entity'de deletedAt alanı var) [128] [129]. Yani block.deletedAt = now, transaction.deletedAt = now gibi. Bu da bir soft delete yöntemi.

Burada kritik olan, ileride yeni bir blok gelirse (yeniden organize edilen) onu zaten normal akış ekleyecek. Biz eskiyi geçersiz yaptık ki raporlarda vs. gözükmesin. Belki purge mekanizması eklenir (çok eski ve invalid kayıtları tamamen sil). Ama ilk aşamada invalid olarak işaretlemek yeter.

Bu geliştirme yapılmazsa, **veri tutarlılığı** bozulacak ve kullanıcı yanlış verilere güvenecektir. Bu nedenle kritik bir güncellemedir.

6. Transaction İşleme Sırasında Kısmi Kayıt (Yarıda Kalma) Riski **– *Risk:*** Yüksek. **Chainhook payload'ı işlenirken, bir blok ve onun bazen onlarca işlemi ve yüzlerce event'i olabilir. Şu anki kodda anladığımız kadarıyla tek bir @Transactional içinde block ve transactions kaydediliyor, event'ler de sonra kaydediliyor [114]. Eğer event'lerin kaydı sırasında bir hata oluşursa (ör. beklenmeyen bir veri veya DB hatası), Spring tüm transaction'ı geri alır, bu iyi. Ancak ya tam tersi olursa: Kod, block ve transaction'ları veritabanına yazıp commit ederse (belki orada ayrı bir transaction boundary var) ama event yazarken hata alırsa? O zaman block ve transaction kalmış, eventler yok, yani veritabanında tutarsız bir durum. Özellikle manual transaction yönetimi yapıldıysa (örn. saveAll çağrıları belki anında commit ediyor, emin olmalıyız). Spring Data JPA genelde tek transaction'da hepsini halleder, ancak garantiye almak iyi olur. *Çözüm:*** Batch işlemlerde parça parça transaction** yönetimi yapmayı öneriyor doküman [130] [131]. Yani 50-100 işlemlik chunk'lar halinde veritabanına yazıp arada commit etmek, hata olursa sadece o chunk'ı rollback etmek. Aslında bu belki reorg olayından bağımsız bir şey; çok büyük payload'larda bellek şişmesini önlemek veya kısmi progress sağlamak için.

İki seviye çözüm var: - Basitçe, bir payload içindeki işlemleri `processChunk` gibi ayrı transactional metodlarla 100'lük gruplara ayırıp kaydedebiliriz (dokümanda örneği var) [130] [131]. Bu sayede bir chunk hata verirse o chunk rollback olur ama önceki chunk'lar commitlenmiş kalır. Bu tabii veride yine tutarsızlık yaratabilir (yarım blok işlemleri var) fakat belki sonraki bir mekanizma ile yeniden denenecektir. Bu ancak hataların nadir olduğu varsayımıyla çalışır.

- Daha güvenilir bir yaklaşım, **Spring Batch** modülünü kullanmak. Spring Batch, büyük veri setlerini commit aralıklarıyla işlemeye uygun bir framework. Örneğin dokümanda bir Job Step pseudo-konfigürasyonu verilmiş [132]: chunk(100) ile okuma/işleme/yazma adımları tanımlanıyor, bazı hatalarda skip ediliyor vs. Bu entegre edilirse, işlemlerin bir kısmı başarısız olsa bile job devam edebilir, raporlama yapabilir. Fakat Spring Batch eklemek MVP için belki fazla gelir. Yine de

enterprise düzeyde düşünürsek, eğer her block processing'i bir job olarak ele alırsak daha izlenebilir ve yönetilebilir olur (Spring Batch'in kendi tabloları vs. var, belki gereksiz karmaşıklık şu an).

Pratik çözüm: **Transactional demarkasyonunu doğru yerleştirmek.** Belki en iyisi, WebhookController çağrısında chainhook payload işlenirken, bu işlem zaten doğal olarak tek bir block veriyor. Her block için tek transaction kullanmak mantıklı, ancak içinde çok kayıt varsa belki Spring otomatik batch insert yapar (Hibernate `hibernate.jdbc.batch_size` ile) bu da bir committe çok insert demek. Bizim asıl kaygımız *hata olursa her şeyi geri al* vs *kısmi kalma*. Bence en temiz yol, olduğu gibi tek transaction bırakmak (zaten Spring defaultPropagation). Eğer herhangi bir kayıtta hata olursa tüm blok işlemeyi rollback edecek. Bunda problem yok aslında: O blok atlanmış olur. Belki chainhook tekrar dener (Chainhook belki garantili teslim etmiyor ama mempool iletilebilir). Bu MVP için kabul edilebilir. İleride, eğer tek transaction performansı düşürüyorsa (çok uzun sürüp timeout vs.) ozaman chunking düşünülür.

Doküman partial commit riskine dikkat çektiği için mention ettik, ancak ben *tutarlılık açısından* tek transaction iyidir diyorum. Dolayısıyla, bu riskin çözümü: - Tüm block işleme tek bir @Transactional içinde olmalı ki ya hep ya hiç. Bunu zaten yapmışlar belki. - Alternatif, yukarıdaki chunk yaklaşımı, ama bu tutarlılığı biraz bozar ama ilerleme sağlar.

Kısacası burada kararı gereksinimler belirleyecek. MVP için belki tek tx kalsın, ileride devasa bloklar gelirse optimize edilir. Yine de mevcut koda göz atıp partial commit riski var mı kontrol edilmeli. Eğer `saveAll` içinde exception olduğunda önceki save'ler commit olduysa tehlike var – ama Spring Data JPA, default olarak transaction içinde olduğu sürece commit anı, method çıkışında olur. O yüzden belki sorun yok. Bu not daha çok farkındalık için.

7. TransactionEvent Veri Modelinin Tasarım Sorunu (Çok Sayıda Nullable Alan) **– *Risk:*** Orta. **Domain modelinde** TransactionEvent **tek bir tablo olarak tanımlanmış ve farklı event tiplerine ait tüm alanlar bu tabloda kolon olarak duruyor. Örneğin FT_TRANSFER tipi için assetIdentifier, amount, sender, recipient kolonları varken, STX_LOCK tipi için lockAmount, unlockHeight kolonları var – ve bunların çoğu diğer tiplerde null kalıyor** [133] [134]**. Bu, veritabanında boş alan israfına ve veri bütünlüğü için zafiyete yol açıyor (yanlış tip için alan dolması vs. mümkündür). Ayrıca uygulama kodunda da her event için büyük bir sınıf olup her yerinde null kontrolü yapmak gerekiyor. *Çözüm:*** JPA Inheritance **kullanarak her event tipini ayrı entity olarak temsil etmek en temiz yol. Zaten hali hazırda TransactionEvent bir base class olabilir, alt sınıflar mesela FTTransferEvent, NFTTransferEvent, StxLockEvent vs. olarak tanımlanabilir. JPA'de üç strateji var, burada** JOINED inheritance** öneriliyor [135] [136] çünkü SINGLE_TABLE yine aynı null problemine yol açar, TABLE_PER_CLASS ise sorgulaması zor ve performanslı değil. JOINED ile her alt sınıf için ayrı tablo olur, ortak alanlar base tabloda, spesifik alanlar kendi tablosunda. Örnek:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name="event_type")
public abstract class TransactionEvent {
    @Id @GeneratedValue Long id;
    @ManyToOne(fetch=LAZY) StacksTransaction transaction;
    @Column Integer eventIndex;
    @Column String contractIdentifier;
    @Column Instant createdAt;
    // ... ortak alanlar
```

```
}

@Entity @DiscriminatorValue("FT_TRANSFER")
public class FTTransferEvent extends TransactionEvent {
    private String assetIdentifier;
    private BigDecimal amount;
    private String sender;
    private String recipient;
    // belki token türü vs.
}

@Entity @DiscriminatorValue("STX_LOCK")
public class StxLockEvent extends TransactionEvent {
    private Long lockAmount;
    private Long unlockHeight;
}
```

Yukarıdaki gibi, her alt tip kendi kolonlarına sahip olur. Uygulamada, ChainhookPayloadParser gelirken eventType'a bakıp ilgili alt sınıf nesnesi oluşturabilir. Polimorfizm sayesinde `transactionEventRepository.findByTransactionId()` gibi sorgular hala tüm alt tipleri getirebilir (JPA buna izin verir, base class üzerinde sorgu yazınca JOIN yapar alt tablolara). Bu tasarım ile veritabanı normalleştirilir, her kolon sadece ilgili event'te dolu olur, veri tutarlılığı artar (örneğin FTTransferEvent'te amount null olamaz gibi constraint koyabiliriz). Tek dezavantaj, event okuma yazma işlemleri bir JOIN gerektirecek, ama bu kayda değer bir overhead değil. Endüstride bu tip hiyerarşik veriler için genelde JOINED tavsiye edilir.

Bu değişiklik, mevcut veritabanı şemasını değiştirir. Flyway ile yeni migration yazıp TransactionEvent tablosunu alt tablolara bölmek gerekir. Bu belki MVP aşamasında hemen yapılmayabilir (tüm koda dokunmak demek). Ancak uzun vadede kod basitleştireceği ve hata riskini azaltacağı için değerlidir.

8. AlertRule Koşullarının JSON olarak tutulması (Tip Güvenliği Yok) – ***Risk:*** Orta-Yüksek. **AlertRule entity'sinde kullanıcı tanımlı koşullar tek bir JSON stringi olarak `conditions` alanında saklanıyor (TEXT kolonu)** [137] **. Bu, esneklik sağlıyor gibi görünse de şöyle problemler var: i)** Tip güvenliği yok – *koşul anahtarları veya değer tipleri tamamen serbest, runtime'da parse edilirken hata olabilir, ii) Bu alana göre sorgu yapmak zor (JSON query yapabilir ama performanslı değil), iii) Kullanıcı hatalı JSON girerse (söz dizimi hatası vb.) bunu sistem yakalamazsa runtime'da patlar, iv) Farklı kural tipleri için farklı JSON şemaları gerek, vs. Çözüm:* İki yönlü düşünülebilir:

- **JSON Schema Validation:** Eğer conditions alanını tutmaya devam edeceksek, en azından JSON formatının doğruluğunu ve beklenen alanları içerip içermediğini kontrol etmeliyiz. Örneğin her kural tipine göre bir JSON şeması tanımlanabilir. Kullanıcı kural oluştururken (AlertRuleController'da) `conditions` string'i, ilgili şemaya karşı doğrulanır (bunu yapan Java kütüphaneleri var, everit JSON Schema gibi). Böylece hatalı input baştan engellenir. Ayrıca Hibernate Validator ile entity üzerinde de @AssertTrue benzeri bir custom annotation yazılabilir ama en kolayı service katmanında kontrol etmek. Hatta dokümanda, JSON'ı direkt Map şeklinde saklamak için Hibernate Types kütüphanesi ve PostgreSQL jsonb kullanılması önerilmiş [138] . Bu da olur; jsonb olunca DB tarafında sorgular bile yazılabilir ama bence tip güvenliği sorununu çözmez, sadece biraz performans kazandırır.

- **Polimorfik AlertRule Hiyerarşisi:** Yukarıda bahsettiğimiz gibi, AlertRule'u da alt tiplere ayırmak bu sorunu kökten çözer. Örneğin `ContractCallAlertRule` sınıfı contractIdentifier, functionName, amountThreshold alanlarına sahip olur – böylece JSON parse'a gerek kalmaz, her kural tipinin parametreleri ayrı kolonda tipli olarak durur [139] [140]. Bu yöntem, koşul kontrolünü de basitleştirir (if'lerle parse etmek yerine direkt alan karşılaştırması yaparız). Dezavantajı, kural tipi eklemek için şema değiştirmek gerekir (ama bu belki çok sık olmayacak bir durum). JOINED stratejiyi burada belki değil de SINGLE_TABLE yapabiliriz, çünkü AlertRule alt tiplerinde genelde az sayıda field var ve bu field'lar eğer null olsa bile çok problem değil (kural sayısı muhtemelen çok fazla olmaz event sayısı kadar). Fakat tutarlılık adına JOINED yapıp her kural tipini ayrı tabloya da koyabiliriz. Bu detay tasarım kararı, ama kod tarafında alt sınıflara bölmek bence JSON string tutmaktan iyidir.

Sonuç olarak, MVP için belki JSON olarak bırakmak hızlı çözüm, ama production'a çıkmadan en azından JSON validasyonunu ekleyelim. Ve kullanıcı arayüzünde (frontend) bunu doğru doldurması için kısıtlar koyulmalı. Orta vadede alt sınıf tasarımına geçmek daha sağlıklı.

9. Veritabanı İndeks Yükü ve İleride Partisyonlama İhtiyacı **– *Risk:*** Orta. **Önceki bölümlerde de bahsettik; TransactionEvent üzerinde 8 indeks var [89]. İndeksler sorguları hızlandırır ama her insert/update'te bu 8 indeks de güncellenir, bu da yazma performansını düşürür. Eğer sistem saniyede onlarca event almaya başlarsa, Postgres'in bu indeksleri güncellemesi birikmeye yol açabilir. Aynı şekilde ContractCall üzerindeki 5 indeks de benzer bir durum. *Çözüm:* Bu risk anında bir sorun yaratmayabilir (şu an load düşükse) ama ileriyi düşünerek: - Gereksiz indeks varsa kaldırılmalı. Örneğin composite index (eventType+contractIdentifier) eklenmiş [141] [142], bunu gerçekten kullanan sorgular olduğu için eklenmiş olabilir. Ama belki bazı kombinasyonlar kullanılmıyordur. EXPLAIN ANALYZE ile sorgu planları incelenip hangilerinin efektif olduğu anlaşılabilir [99]. -** Partial Index: **Sık kullanılan değerler için kısmi indeks mantıklı. Dokümanda örnek vermiş: sadece event_type = 'FT_TRANSFER' için assetIdentifier üzerinde indeks oluşturmak gibi [100]. Bu sayede mesela diğer tipler bu indeksin maliyetine katlanmaz. Partially index ile bellek kullanımını da azaltırsınız. -** Partitioning: **Bu ölçeklenebilirlik hamlesi, özellikle** büyüyen veri **için önemlidir. TransactionEvent tablosu her gün 100k büyürse bir yıl sonra 36 milyon kayıt eder. Sorgular yavaşlar, indeksler şişer. PostgreSQL 12+ sürümünde native partitioning iyi çalışıyor. Öneri, zaman bazlı partisyon yapmak [143] [144]. Örneğin her ay için ayrı bir partition tablo. Böylece sorgular son X aya bakıyorsa sadece ilgili partition'a gider. Eski veriyi de READ ONLY yapıp arşivleyebilirsiniz. Hatta TimescaleDB gibi zaman serisi odaklı bir eklenti de düşünülebilir [145] [146]. -** Tuning: **Veritabanı tarafında vacuum, fillfactor gibi parametrelerle indeks overhead'i azaltılabilir. Ancak asıl nokta, bu tablolara gelen sorguları optimize etmektir. Mesela alert matching çoğunlukla eventType+contractIdentifier ile yapılıyor denmiş – eğer biz bellek tarafında eşleştirme yaparsak belki veritabanı sorgusu bile yapmayız. O zaman bu indeksler bir nebze önemsizleşir (çünkü kural eşleştirmek için DB'ye gitmeyeceğiz, zaten event verisini ekledik, kural eşleşmesini uygulama yapıyor). -** Yazma Optimizasyonu: **Eğer insert işlemleri birikirse, belki** bulk insert** veya asenkron insert mekanizmaları (copy komutu vs.) değerlendirilebilir. Spring Data JPA zaten `saveAll(List)` ile toplu insert yapabilir (JPA property ayarı ile). Bu, tek tek insert yerine 100'lü batch atar, bu da indeks güncellemelerini gruplayabilir.

Özetle, indeks konusunu şimdilik not etmek ama büyük yük yoksa aynen bırakmak da mümkün. Fakat production öncesi DB uzmanıyla bu indekslerin gerekliliği ve verimliliği gözden geçirilmeli. Gerekiyorsa yukarıdaki yöntemlerle iyileştirilmeli.

10. Observability Eksikleri (Metrikler, Logging, Health Checks) **– *Risk:*** Orta. *Sistem hatalarını çözmek veya performansı izlemek için gözlemlenebilirlik şart demiştik. Şu an custom metrics tanımlı değil, health endpoint var ama belki yüzeysel, tracing yok. Çözüm:*

- **Uygulamaya Özgü Metrikler:** Micrometer ile sayıcılar, ölçerler ekleyelim. Örneğin her işlenen Chainhook payload'ı için bir counter arttırılabilir [147], her tetiklenen alert için ayrı bir metric tutulabilir, kural eşleştirme süresini ölçmek için Timer kullanabiliriz. Hatta aktif kural sayısı, kuyrukta bekleyen bildirim sayısı gibi Gauge'lar koyabiliriz [148] [117]. Dokümanda verilen örnek:

```
@Component
public class ChainhookMetrics {
    private final Counter processedPayloads;
    private final Timer processingTime;
    private final Gauge activeRuleCount;
    @Autowired
    public ChainhookMetrics(MeterRegistry registry, AlertRuleRepository
ruleRepo) {
        this.processedPayloads =
Counter.builder("chainhook.payload.processed")
            .description("Number of Chainhook payloads
processed").register(registry);
        this.processingTime = Timer.builder("chainhook.processing.time")
            .description("Time taken to process a Chainhook
payload").register(registry);
        this.activeRuleCount = Gauge.builder("alert.rules.active",
            () -> ruleRepo.countActive())
            .description("Active alert rule count").register(registry);
    }
    public void countPayloadProcessed() {
        processedPayloads.increment();
    }
    public <T> T recordProcessingTime(Supplier<T> supplier) {
        return processingTime.record(supplier);
    }
}
```

Bu kabaca bir metric komponenti. UseCase içinde `metrics.recordProcessingTime(() -> processPayload(payload))` gibi çağrılırsa, otomatik süresini ölçer. Sonra Prometheus endpoint'inden bu metrikler toplanabilir.

- **Actuator Endpoints:** */actuator/health*, */metrics*, */prometheus* gibi endpoint'ler açık olmalı. Zaten config'de prometheus enable var görünüyor [149]. Health'in belki özelleştirilmesi gerekebilir: ör. veritabanı bağlantısı, bellek durumu, vs. health indikatörleri eklendi mi bakılmalı. Spring Boot çoğunu otomatik yapar.

- **Logging & Tracing:** Tüm kritik aksiyonlarda anlamlı log'lar basılması lazım. Örneğin yeni bir block geldiğinde "INFO: Processed block X with Y transactions", bir kural tetiklendiğinde "INFO: AlertRule #5 triggered for tx 0xabc...", bir hata oluşursa ilgili detaylar (mümkünse stack trace ile) "ERROR: Failed to send email to user X". Şu an global exception handler var, muhtemelen validation hatalarını vs. HTTP 400 olarak dönüyor, bu iyi. Ama biz arka plandaki hataları da

yakalayıp loglamalıyız. Async method içinde exception fırlarsa ne oluyor? Spring, @Async method hatalarını handle edemeyebilir (SimpleAsyncUncaughtExceptionHandler ile log basılabilir). Bunu test etmek lazım.

Yapılabilecek iyileştirmeler: - Log formatını JSON yapmak (logback-spring.xml'de pattern ayarı ile). Bu sayede Kibana gibi araçlar logları rahat parse eder. - Her request'e bir trace id (Sleuth kullanınca otomatik geliyor) eklemek, log'larda o id'yi basmak. Böylece bir isteğin tüm loglarını zincirlemek mümkün olur. - *Distributed Tracing:* Şu an belki gereksiz, ama eğer mesela Chainhook -> Bizim servis -> başka servis şeklinde bir zincir olursa (ya da ileride microservice yapılırsa) Zipkin/Jaeger ile entegre olmalı. Spring Cloud Sleuth 3.x artık Micrometer Tracing ile entegre, onu inceleyip ekleyebiliriz. - **Healthchecks:** Actuator health zaten veri tabanı vs. kontrol ediyor. Ek olarak belki chainhook için de bir health indikatörü koyulabilir (ör. "son 10 dakikadır yeni block gelmedi, belki sorun var" gibi – bu daha ileri seviye). - **Cache Gözlemlenebilirliği:** Ehcache veya Caffeine vs. kullanılsaydı Actuator metric gelirdi. Şu an kendi cache implementasyonumuz var. Onu belki metric ile gauge'ladık (activeRuleCount gibi). Alternatif, Spring Cache abstraction kullanılabilirdi (starter-cache ekli, belki kullanılmıştır). Örneğin `@Cacheable("activeRules")` tarzı, ve `@CacheEvict` vs. Bu da düşünülebilir. - **Profiling:** Production öncesi bir JProfiler ile test ortamında bakılabilir, en çok zaman alan yerler vs. tespit edilebilir. Bu proaktif bir adım olur.

Orta Düzey (Medium) Eksikler / İyileştirmeler

(Not: Bazı orta düzey maddeler yukarıda kritiklerde iç içe anlatıldı, burada önemli olanları derleyip kısaca geçiyoruz.)

- **Veri Önbellekleme (Caching) İyileştirmeleri:** Uygulamada Spring Cache altyapısı eklenmiş fakat aktif kullanımı sınırlı olabilir. Potansiyel olarak, sık çağrılan read-only sorgular (ör. son blok yüksekliği, sık sorulan transaction aramaları) cache'lenebilir. Örneğin `BlockQueryService.getLatestBlockHeight()` varsa mem-cache'ten dönebilir, veya AlertRule'ların DB'den okunması zaten memory'de tutuluyor olabilir. Eğer sistem tek instance kalacaksa Caffeine gibi gömülü bir cache kullanılabilir; cluster'da ise Redis'e geçmek gerekir. Şu an **caching** hayati bir eksik değil ama performansı artıracak düşük asılı meyve (low hanging fruit) diyebiliriz. Özellikle **AlertRule caching** zaten yapılmış dedik; onun dışında belki *MonitoredContract listeleri* vs. de cache'e alınabilir. Sonuç: Orta öncelikli bir geliştirme.

- **API Anahtar Yönetimi:** Harici servislerin API'ları kullanabilmesi için her kullanıcıya (veya projeye) bir API key verilebilir. Bu key ile limitli erişim, ip kısıtı vb. konulabilir. Bu proje doğrudan böyle bir ihtiyaçtan bahsetmiyor ama gelecekte üçüncü parti entegrasyonlar istenirse gündeme gelebilir. Bunu not olarak düşüyoruz.

- **Kullanıcı & İzin Sistemi Geliştirmeleri:** Şu an kullanıcılar sadece kendi alert kural ve bildirimlerini görür farz ediyoruz. Multi-tenant bir sistem gibi. Bunu enforce etmek için, tüm repository sorgularında userId kriteri kullanılmalı (AlertRuleRepository.findByUserId vs. varmış 150). Bu mantık her yere yerleştirilmiş mi kontrol edilmeli. Admin rolü gelirse o tüm kayıtları görebilmeli vs. Bu detaylar orta seviyede planlanmalı.

- **Dökümantasyon ve API versiyonlama:** Son kullanıcıya dönük bir doküman olmamakla birlikte, geliştirici dokümantasyonu (README, API docs) güncel olmalı. Örneğin *OpenAPI (Swagger)* eklenip API'lar otomatik dokümante edilebilir. Bu, dış paydaşlara API sunarken endüstri standardıdır. Versiyonlama konusunda, halihazırda /api/v1 prefix kullanılıyor, bu iyi. İleride breaking change olursa v2 eklenir vs.

- **Deployment Hazırlıkları:** Production ortamda container kullanmak fiili standart. Bu projeyi **Dockerize** etmek gerekiyor. Muhtemelen basit bir Dockerfile ile başlayabilir:

```
FROM eclipse-temurin:17-jdk-alpine
WORKDIR /app
COPY target/stacks-monitor.jar .
EXPOSE 8080
ENTRYPOINT ["java","-jar","stacks-monitor.jar","--
spring.profiles.active=prod"]
```

gibi. Sonra docker compose ile Postgres ve uygulama tanımlanır. Kubernetese gidecekse orada ConfigMap/Secret ile chainhook secret, DB creds vs. yönetilir. Bunlar proje kodundan ziyade devops konuları ama mimari olarak externalize config mantığı (Spring Boot already does via application-prod.yml) güdülmüş.

- **Testlerin Artırılması:** Orta öncelikte fakat unutulmamalı; özellikle kritik akışlar için test yazmak lazım demiştik. JUnit + Mockito ile ünit testler, Testcontainers ile entegre testler, belki Gatling veya JMeter ile load test (20k event ne kadar sürede işleniyor görebiliriz).

Düşük Öncelikli / Kolay İyileştirmeler

- **Log İyileştirmeleri:** Yukarıda bahsettik, JSON logging, korelasyon ID gibi eklemeler. Uygulaması nispeten kolay (logback konfigürasyonu değişikliği ve belki %X{traceId} pattern kullanımı). Seviyelerin doğru ayarlanması vs.

- **Küçük Güvenlik İyileştirmeleri:** Content Security Policy, CORS kısıtlamaları (eğer frontend ayrı domain'de olacaksa), HTTP header sertleştirmeleri vs. yapılabilir. Bunlar Spring Security'de kolayca eklenir (`.headers().contentSecurityPolicy("default-src 'self'")` vs.). Bu seviye belki şu an gereksiz ama production checklist'inde olmalı.

- **Kullanılmayan Kodların Temizliği:** Örneğin belki boşa çıkan repository metotları veya eskiden düşünülmüş ama implemente edilmemiş (mesela RateLimitFilter var ama devre dışıysa) kodlar temizlenmeli, böylece kafa karışıklığı olmaz. Yine configuration'da kullanılmayan property'ler vs. ayıklanabilir.

- **Refactoring için Alanlar:** Kodun genel yapısı iyi olmakla birlikte belki daha tutarlı adlandırma veya sınıf sorumluluğu ayrımı gerekebilir. Örneğin *MonitoringController* ve *MonitoringService* var, belki bunlar yerine Actuator kullanılsa gerek kalmaz ama bu minör bir konu. Veya *AlertMatchingService* vs *EvaluateAlertRulesUseCase* – belki bir naming standard oturtulabilir (service mi usecase mi). Bu gibi refactorlar teknik borç olarak görülebilir.

- **UI/UX Hazırlıkları:** Frontend kapsam dışı dediniz ama backend'de belki bir Swagger UI koymak (springdoc-openapi Ui ile) developer UX'i için iyi olur. Yine, belki basit bir admin UI'si yapılıp release edilebilir, ama bu sorunun kapsamı değil.

Yukarıdaki tüm eksik/risk alanlarını özetlersek: **Kritik** olanlar güvenlik, kural motoru performansı, notification retry ve reorg handling konuları. **Orta** olanlar veri modeli optimizasyonları (inheritance), JSON tip güvenliği, indeks/partisyon kararları, observability, caching vb. **Kolay** kategorisinde ise log ve dokümantasyon gibi ufak iyileştirmeler yer alıyor. Bu önceliklendirmeye göre hareket etmek, kaynakları verimli kullanmak açısından önemli.

Aşağıdaki bölümde, bu iyileştirmelerin **MVP geliştirme planına** nasıl yansıtılacağını ve adım adım ne şekilde ele alınacağını belirteceğiz.

MVP Geliştirme Planı ve Yol Haritası

Yukarıdaki analiz ışığında, sistemin production-ready bir MVP olarak hayata geçirilmesi için yapılması gereken işleri mantıklı fazlara ayırdık. Bu yol haritasında her faz (P0, P1, P2) belirli bir odak alanına yöneliktir. Her faz içinde, Claude Code gibi bir yardımcıya kodlatabileceğiniz somut görevler sıralanmıştır. **Öncelikli amaç**, önce sistemi güvenli ve temel fonksiyonları çalışır hale getirmek (P0), sonra performans ve işlevsellik optimizasyonları (P1) ve son olarak operasyonel mükemmellik ve ölçeklenebilirlik (P2) adımlarını tamamlamaktır.

Faz P0 – Çekirdek Sistem ve Güvenlik Altyapısı

Amaç: Sistemin temel domain modelini ve kritik altyapı servislerini ayağa kaldırmak. Bu aşamada sistem "ucundan uca" çalışabilir hale gelmeli (blok verisini alıp basit kural tetikleyip bildirim gönderebilmeli) ancak gelişmiş optimizasyonlar ve ikincil özellikler sonraya bırakılacak. Güvenlik temelleri de bu fazda atılacak ki sonraki geliştirmeler güvenli bir ortamda ilerlesin.

- **Domain Modellerinin Oluşturulması:** Öncelikle **core entity** sınıflarını ve JPA mapping'lerini oluşturun. Bunlar: `StacksBlock`, `StacksTransaction`, `TransactionEvent` (ve alt tipleri veya mevcut hali), `ContractCall`, `ContractDeployment`, `MonitoredContract`, `AlertRule` (ve alt tipleri/polimorfik yapı veya şimdilik JSON field'lı haliyle), `AlertNotification`, `User` gibi sınıflardır. İlişkileri ve veritabanı kolonlarını doğru şekilde tanımlayın (ör. Transaction → Block many-to-one, TransactionEvent → Transaction many-to-one, vs.). Gerekirse başlangıçta TransactionEvent'i basit tutup alt tip işini P1'e de bırakabilirsiniz, ama şema değişikliği olacağı için belki en başta yapıp kurtulmak daha iyi. Aynı şekilde AlertRule alt tiplerini de ilk seferde tasarlamak mantıklı olabilir. Claude Code'a bu entity sınıflarını JPA anotasyonlarıyla birlikte yazdırabilirsiniz. Ardından **Flyway migration** dosyasını yazdırın (V1__initial_schema.sql) – aslında elinizde bir tane var [61], belki onu güncelleyeceksiniz yeni yapıya göre. Bu adım tamamlandığında veritabanı modeli hazır olacak.

- **Repository Arayüzlerinin Yazılması:** Her bir entity için Spring Data JPA repository arayüzlerini tanımlayın (StacksBlockRepository, StacksTransactionRepository, vb. mevcut kodda var zaten [151] [152]). Claude Code'a bunları metod imzalarıyla ürettirebilirsiniz. Önemli özel sorgular varsa (örn. findByBlockHash, findByUserIdAndStatus vs.), bunları method name ile ya da @Query ile tanımlayın. Bu sayede veri erişim katmanı hazır olacak.

- **Service ve UseCase Sınıflarının İskeleti:** Uygulama katmanı için, temel akışı yönetecek sınıfları oluşturun. En kritikleri:

- `ProcessChainhookPayloadUseCase` – WebhookController'dan çağrılacak, gelen Chainhook payload'ını parse edip block/tx/event kayıtlarını yapan ve AlertMatching'i tetikleyen use-case. Claude Code'dan, JSON DTO nesnelerini parametre alıp domain nesneleri oluşturan iskelet kodu yazmasını isteyin. (Parse işini belki ayrı Parser sınıfına bırakacak).
- `AlertMatchingService` – Bir transaction veya event listesini alıp alert rule'ları değerlendiren servis. Şimdilik basit implementasyon (tüm aktif kuralları döngüyle kontrol) yapabilir, P1'de optimize edeceğiz. Yine de metod imzasını (`evaluateTransaction(StacksTransaction tx)` gibi) belirleyip belki basit bir if/else ile kural tetikleme logiğini kurun ki uçtan uca çalışsın.

- `NotificationDispatcher` – Yeni tetiklenen AlertNotification'ı alıp ilgili kanal servisine veren servis. Bunu belki basit tutup, direkt senkron çağırabilirsiniz P0'da (async ve retry P1'de). Ama e-posta gönderimi zaman alabilir, belki en baştan @Async kullanmak zararsız. Karar size kalmış; minik bir thread pool config ile yapabilirsiniz.
- `EmailNotificationService` ve `WebhookNotificationService` – bunlar NotificationService arayüzünü implemente eden somut sınıflar olsun. İçlerinde şimdilik dummy `send()` yapabilirsiniz (örn. konsola "Email sent to X" basabilir) ya da gerçekten mail atmak isterseniz Spring Boot Starter Mail'i konfigure edip JavaMailSender kullanın. P0 için belki gerçek mail entegrasyonu yapılabilir çünkü çok karmaşık değil (application.yml'de SMTP ayarları gerekebilir). Webhook için RestTemplate ile bir dummy siteye POST atabilirsiniz test için.

- `AuthenticationService` – register ve login işlemlerini yapan servis (yukarıda kod örneğini verdik). Bunu da P0'da halledin ki JWT üretimi çalışsın.

- **Controller'ların Oluşturulması:** Sunum katmanında, gerekli tüm controller endpoint'lerini oluşturun. En önemlisi `WebhookController` (POST /webhook/chainhook). Bu, gelen JSON'u alıp hemen ProcessChainhookUseCase'i çağırmalı. P0'da belki Chainhook'tan gerçek veri gelmeyeceği için manuel test edersiniz. Ayrıca `AuthController` (register, login) ekleyin. AlertRuleController, TransactionQueryController gibi şeyler P0 için olmazsa da olur (kullanıcı arayüzü yoksa). Ama belki minimum bir AlertRuleController (kural listeleme vs.) eklerseniz Postman ile test kolaylaşır. P0'da hedef, dış dünyadan *en az* chainhook post'unu alıp bir iki dummy kuralla e-posta log basana kadar tüm zinciri test etmek olduğundan, belki kural eklemek için ya veritabanına manuel insert ya da hızlıca bir POST /alerts/rules eklersiniz. Bu ayrıntılar geliştirme sırasında netleşir.

- **Güvenlik Katmanının Entegrasyonu:** Yukarıdaki kritik eksik 1 ve 2 maddelerindeki JWT, SecurityConfig, HMAC filter konularını P0'da halledin. Çünkü sisteme bundan sonra eklenecek her şey, güvenli bir şekilde çalışmalı. Yani:

- JWT token üretimini login akışına ekleyin.
- JwtAuthenticationFilter ve SecurityConfig ile tüm /api/** çağrılarını koruma altına alın (webhook ve auth hariç) [81] .
- ChainhookHmacFilter'ı da SecurityConfig'e veya FilterChain'e ekleyin ki ilk günden veri güvenliği olsun.
- Parola hashing vs. gibi konular da P0'da bitsin.

Bu noktada *Claude Code'a SecurityConfig ve filtreleri yazdırmak iyi bir fikir* (yukarıdaki örneği zaten kullanabilirsiniz). Ayrıca PasswordEncoder bean ve WebSecurityConfigurer beans yazdırın.

- **Temel Doğrulama ve Hata Yönetimi:** P0'da belki çok detayına girmeyebilirsiniz ama Spring Validation (@Valid) kullanarak DTO'ların (register, login, add rule vs.) basit alan kontrollerini yapın. GlobalExceptionHandler zaten var ise onu kullanın, yoksa yazdırın (mesela MethodArgumentNotValidException vs. yakalayıp 400 dönsün). Bu sayede temel input validation sağlanır. Bu adım küçük ama kaliteli bir API için önemlidir, o yüzden P0'da yapılabilir.

- **İlk Uçtan Uca Test:** P0 sonunda, tüm parçaların entegre çalıştığını doğrulayın. Bunun için:

- Test ortamında uygulamayı çalıştırın.
- Chainhook endpoint'ine eldeki bir örnek JSON payload'ını POST edin (bunu dokümanlardan veya test datalarından alabilirsiniz). Bu request HMAC imzası gerektireceği için header'a uygun imza

koymayı unutmayın; test için filtreyi devre dışı bırakabilir veya secret'ı vs. bilerek hesaplayabilirsiniz.
- Sistem payload'ı alıp veritabanına block/transaction/event verilerini ekledi mi bakın (DB'ye query).
- Sonra event'e uygun dummy bir kural ekleyip (manuel olarak AlertRule tablosuna insert belki) aynı payload'ı tekrar yollayın, bu sefer kural tetiklenip AlertNotification oluşuyor mu gözlemleyin.
- NotificationDispatcher bunu alıp EmailNotificationService.send çağırdı mı (loglardan veya debug ile bakın).
- Bütün bu akışta bir yerlerde hata varsa düzeltin.

Bu test başarılı ise, çekirdek sistem çalışıyor demektir. P0 deliverable olarak elde güvenli, temel fonksiyonları yerine getiren bir backend olacak.

Faz P1 – İşlevsellik Tamamlama ve Performans Optimizasyonu

Amaç: P1 aşamasında sistemin tüm önemli işlevleri eksiksiz hale getirilecek ve P0'da basit bırakılan kısımlar optimize edilecektir. Özellikle **alert rule motoru**, **bildirim mekanizması**, **veri modeli** gibi kısımlar bu fazda güçlendirilecek. Ayrıca P0'da ihmal edilen bazı güvenlik/araç konuları da burada ele alınacak.

- **Alert Rule Motorunun İyileştirilmesi:** P0'da basit bir tümünü dolaş mekanizması vardı. P1'de bunu, yukarıda detaylandırılan **önbellek ve indeksleme stratejisi** ile geliştiriyoruz. Claude Code'a `AlertRuleCacheService` gibi bir sınıf yazdırın (ya da mevcut AlertMatchingService içine de gömülebilir). Tüm aktif kuralları belli aralıklarla yükleyip sınıflandırma (contractId → kurallar listesi, eventType → kurallar listesi) yapacak şekilde implement edin [101] [153]. Sonra AlertMatchingService.evaluateTransaction() metodunu, önbellek üzerinden ilgili kural alt kümesini alacak ve sadece onları döngüleyecek şekilde değiştirin. Bu, performans artışı sağlayacak.

Ayrıca, **polimorfik AlertRule** yapısını devreye sokmak isterseniz (belki P0'da yapmamıştınız): Örneğin JSON conditions yerine alt sınıflar (ContractCallAlertRule vs.) oluşturup, evaluate kısmını her alt sınıfın kendi `matches(event)` metoduna bırakabilirsiniz. Bunu yapmak kodu sadeleştirir. Ancak DB tarafı için migration gerektirir (discriminator column ekleme vs.). Eğer P0'da JSON ile bıraktıysanız, P1'de bunu değiştirmek biraz iş olabilir (veri dönüşümü vs.). Bu nedenle belki en başta yapmış olmayı tercih edersiniz. P1'de de yapabilirsiniz ama dikkatli planlayın.

Ek olarak, *cooldown mekanizması* kontrol edin: Muhtemelen AlertRule.lastTriggeredAt alanı var ve her tetiklenmede update ediliyor. Bu update işleminin transaction içinde yapılması iyi olur ki race condition olmasın (aynı anda iki event aynı kuralı tetiklerse biri diğerini engellemek için). Şu an sistem tek thread çalıştığı için sorun yok ama multi-thread veya multi-instance olursa burada kilit gerekli olabilir. Bunu belki şimdilik ihmal edebilirsiniz, ama not edin.

- **Notification Sistemi ve Retry Mekanizmasının Eklenmesi:** P1'de, P0'da temel olarak çalışan bildirim gönderimini production seviyesine yaklaştıracağız. Yani:
- **Retry & Circuit Breaker:** Spring Retry ve Resilience4j entegrasyonunu, EmailNotificationService ve WebhookNotificationService'de uygulayın [23] [107]. Kod örneğini Claude Code'a verip kendi projenize uyarlamasını sağlayın. Özellikle Webhook için RestTemplate hatalarını yakalamak için exception türlerini doğru seçin. Email gönderimi için de mail sunucusu hatalarında benzerini yapın (MailException vs.).
- **Dead Letter Queue Mekanizması:** Basit bir çözüm için, yeni bir JPA entity oluşturabilirsiniz: `FailedNotification` mesela, id, notificationId, failReason, failTime gibi alanları olsun.

fallbackMethod'da bunu DB'ye yazsın. İleride belki bir admin panel ile bu tablo izlenir ve tekrar göndermeye çalış butonu vs. yapılır. Daha sofistike bir çözüm için RabbitMQ entegrasyonu yapabilirsiniz. Fakat bir dış servis daha (ve operasyonel yük) ekleyecek, MVP için şart değil. Kod basitliği için DB tablosu yeter.

- **Asenkron İşlem Yönetimi:** P0'da belki NotificationDispatcher zaten @Async idi. Ama hatırlatmak gerekir, @Async kullanıyorsanız bir ThreadPool konfigüre edin (ThreadPoolTaskExecutor bean). Varsayılan thread sayısı vs. ayarlanabilir. Notification yoğunluğunda thread aç/kapat overhead olmasın diye belki corePoolSize=5, maxPool=20 falan belirlenebilir.
- **Batch Gönderim:** Eğer aynı anda çok sayıda bildirim oluşursa bunları toplu işlemek isteyebilirsiniz. NotificationDispatcher'da `dispatchBatch(List<Notification>)` gibi bir metod var [72]. Onu kullanarak belki cron ile her dakika PENDING bildirimleri toplayıp sırayla ateşleyebilirsiniz. Bu pull modeli, event-driven push modeline alternatif. Hangisini kullanacağınız tasarım tercihi. Push (yani her event gelince anında tetikle) şu an var. Ek olarak "retryFailedNotifications" gibi bir periodic job koymak gerekir, belki bir @Scheduled yazıp FAILED durumdakileri tekrar deneyebilirsiniz X kez. Bunu P1'de düşünün: Örneğin 5 dakika ara ile 3 kez dene, yine olmazsa FAILED bırak.
- **Bildirim İçerik ve Şablonları:** Şu an bildirimler muhtemelen sadece basit bir mesaj içeriyor (AlertNotification.message). Bunu oluştururken kullanıcıya gidecek anlamlı bir metin üretmek gerek. Örneğin: "Alert: Contract X function Y called with amount Z > threshold" gibi. Bu belki i18n falan gerektirmez şu an. Ama en azından e-postanın body'sini, konu satırını falan güzel yapın. Spring Boot Mail'de basit text gönderebilirsiniz, isterseniz Thymeleaf template entegre edebilirsiniz (çok ekstraya gerek yok MVP'de).

- **Webhook Payload Format:** Eğer kullanıcıya webhook gönderiyorsanız, onlara ne gönderiyorsunuz? Belki AlertNotification ID, rule name, triggeredAt vs. içeren bir JSON. Bunu tasarlayın ve dökümanlayın ki kullanıcı kendi sistemine entegre edebilsin. Bu da bir contract sonuçta. Şu an sabit bir şey olabilir.

- **Veri Modeli Güncellemeleri:** P1'de planlanmışsa:

- **TransactionEvent alt tip tabloları** migration'ını yapın (eğer P0'da yapmadıysanız). Bu, kodda entity değişikliği + Flyway V2 migration demek. Dikkatli test edin.
- **AlertRule alt tipleri** keza öyle.
- Bu değişiklikler sonrasında repository'lerin de alt sınıflarla çalışması (ya da base class ile bırakabilirsiniz repository'leri, JPA kendisi halleder).
- Ayrıca `AlertNotification` içinde belki eventId tutuyor (ilişki olarak) [154]. Event alt sınıfa bölünce bu foreign key gene base classa referans olur, problem yok.
- `User` entity'ye belki yeni alanlar eklenebilir P1'de (API key, isActive vs.). Bunlar varsa önceden karar verin.

- Veritabanı constraint'lerini gözden geçirin: Örneğin AlertRule.user_id not null olmalı, AlertNotification.alert_rule_id not null vs. Bu gibi veri bütünlüğü kısıtları (foreign keyler, unique indexler) Flyway script'lerine eklenmeli. P0'da belki kabaca oluşturduk, P1'de detaylandırırız.

- **Arama ve Filtreleme API'lerinin Tamamlanması:** Şu an bazı QueryController'lar var (BlockQueryController, TransactionQueryController) [155] [156]. Bunlar blok veya işlem geçmişini sorgulamak için. P1'de bunları bitirebilirsiniz. Örneğin "GET /blocks/latest/height" vs. halledin. Zor bir şey değil, repository'den veriyi çekip dönecek. Hatta caching burada devreye girebilir (latest block number cache'de tutulabilir). Bu API'ler müşteri tarafında gerekli ise önemli, yoksa çok zaman harcamayın. Ama *MonitoringController*'da stats'lar var, belki /stats/blockchain vs. Bu

istatistikleri de doldurmak gerekebilir (büyük ihtimalle basit count query'ler vs.). P1'de bunu yapıp, belki Prometheus metriclerine paralel, API üzerinden de bazı metrik vermiş olursunuz.

- **Rate Limiting Uygulaması:** Yukarıda Bucket4j ile örnek verdik. P1'de bu filtreyi sisteme ekleyin. Çünkü P0'da belki koymamıştık (test ederken zor olmasın diye). Şimdi güvenli tarafta olduğumuza göre, istekleri kısıtlamaya başlayabiliriz. Bu, herhangi bir fonksiyona etki etmez, sadece güvenlik katmanını tamamlar. Aynı zamanda, audit logging de ekleyebilirsiniz P1'de – mesela GlobalExceptionHandler'daki 401, 403 durumlarını loglamak veya RateLimit aşımlarını loglamak gibi.

- **Gözlemlenebilirlik Eklemeleri:** P1 sonunda sisteme dair temel metrikleri ve monitoring araçlarını entegre etmiş olalım. Bunun için:

- Yazdığınız Micrometer metric kodlarını devreye alın (yukarıda ChainhookMetrics vs. örnek).
- Prometheus endpoint'ini test edin, metric'ler geliyor mu bakın.
- Logging'i JSON formatına çevirebilirsiniz bu aşamada (çünkü artık sistem oturdu, logları bozsak da debug edebiliriz).
- Eğer mümkünse, basit bir Grafana kurulumu yapıp metrikleri görmeyi deneyin (bu, kodun parçası değil tabii).

- Hatırlatma: Logging demişken, belki istisna loglarında stack trace basmak önemli. Logback default hata loglarında basar ama eğer global handler'da yakalayıp Response dönüyorsanız, yine de logger.error ile e.printStackTrace() geçin ki kaybolmasın.

- **Test ve Düzeltmeler:** P1 sonunda, P0'da yaptığınız testleri tekrarlayın. Ayrıca ek senaryolar test edin:

- Birden fazla kural senaryosu (iki farklı kural ekleyip aynı event'te ikisi tetikleniyor mu? Biri cooldown'da ise tetiklenmiyor mu?).
- Retry mekanizması testi (örneğin webhook URL'si yanlışsa 3 kez denedi mi? Sonra DB'ye failed notification yazdı mı?).
- JWT koruması testi (token olmadan kural listesi çekilemediğini doğrulayın, token ile başarılı).
- HMAC testi (yanlış imza ile chainhook gönderince 403 alıyor musunuz?).
- Performans testi mini (100 kural ekleyip 1 event gönderin, ne kadar sürede işliyor loglardan ölçün, eğer cache yokken vs. cache varken fark görebilirsiniz).
- Reorg testi (bunu simüle etmek zor belki ama chainhook rollback JSON'ı elle gönderip handleRollback'in çalışmasını deneyebilirsiniz).

Hatalar veya eksikler çıkarsa giderin. P1 bitiminde, sistem işlevsel olarak tam, güvenlikli ve optimize hale gelmiş olacak.

Faz P2 – Prod Ortamı Hazırlığı ve İleri Seviye İyileştirmeler

Amaç: P2, sistemi gerçekten **production-ready** kılan son dokunuşların yapıldığı aşamadır. Bu fazda, geliştirme ortamında stable olan uygulamanın operasyonel ortama alınması için gerekli altyapısal düzenlemeler ve ileri seviye iyileştirmeler yapılır. Bu içerir: dağıtıma hazırlık (containerization, CI/CD), ölçeklenebilirlik için ekstra önlemler, ileri gözlemlenebilirlik, güvenlik sertleştirmeleri, ve gelecek büyümeye hazırlık adımları.

- **Containerization ve Deployment Pipelines:** Uygulama için Dockerfile ve docker-compose hazırlayın. Claude Code'a basit bir Dockerfile yazdırabilirsiniz (Eclipse Temurin Java 17 imajı ile).

Bu imajı build edip, Postgres ile birlikte docker-compose'da ayağa kaldırmayı test edin. Ardından, CI/CD açısından belki GitHub Actions veya Jenkins pipeline ile her push'ta imaj build edip registry'ye (Docker Hub vs.) atma sürecini planlayın. Kubernetes kullanacaksanız, deployment YAML'ları, service, configmap vs. yazılması gerekecek – bunlar devops işleri ama mimari olarak uygulamanın external config okuması vs. hazır olduğu için sorun çıkmaz.

- **Konfigürasyon Yönetimi:** Prod ve test config'lerini ayırın. `application-prod.yml` zaten var [157], ona gerekli prod ayarlarını girin (ör. gerçek DB URL, email SMTP credentials, chainhook secret vs. hepsi environment'tan alacak şekilde ayarlandı mı kontrol edin). Sensitive config'leri (şifreler, secret'lar) asla repo'ya düz metin koymayın; Spring Boot bunları environment variable ile alabilir (`${VAR_NAME}` şeklinde). Bu hazırlığı yapın.

- **Dış Servis Entegrasyonlarını Sağlama:** Prod'da gerçek bir e-posta servisine bağlanacaksanız (SMTP server veya bir servis API'si), onunla entegrasyonu test edin. Gerekirse TLS ayarları vs. ekleyin. Webhook gönderimlerinde belki self-signed sertifika vs. sorun olabilir, bunlara dikkat edin (RestTemplate builder ile .setSSLContext vs. yapmak gerekebilir özel durumlarda). Bu aşamada her şeyin gerçek ortamda çalışacağı düşünülerek ayarlanması lazım.

- **Yatay Ölçek İçin Hazırlık:** Uygulamanın birden fazla instance çalıştırıldığında sorun çıkarmaması gerekir. Stateless olduğu için çoğunlukla hazır, ancak dikkat:

- **Cache**: Eğer AlertRule cache'ini local memory'de tutuyor ve instance başına yapıyorsak, kural eklenince diğer instance'ların haberi olmaz. Basit çözüm: AlertRuleService.createRule gibi metodlarda, veritabanına kaydettikten sonra diğer instance'ların cache'ini temizlemek için bir mekanizma lazım. Bu genelde dağıtık cache kullanmakla çözülür. P2'de isterseniz Redis'e geçin. Spring Cache'i Redis ile konfigure ederseniz (cacheManager bean Redis'i kullanır), tüm instance'lar ortak cache kullanır, invalidation da otomatik olur. Bu biraz ekstra iş ama production cluster için önemli. Alternatif, basit bir yaklaşım olarak "kural eklendiğinde DB'ye yaz, *ardından tüm instance'lara bir WebSocket mesajı at cache temizlesin*" gibi hack de yapılabilir ama profesyonelce değil. En kolayı: **Redis kullanın.** Zaten Redis belki ileride başka amaçlar için de (session store, message broker vs.) kullanılabilir.
- **Singleton İşler:** NotificationDispatcher belki @Scheduled ile periyodik kontrol yapıyorsa, birden fazla instance olunca hepsi yapar, duplication olur. Bunu önlemek için ya böyle işler tek instance'da çalışacak şekilde yapılmalı (kubernetes'de leader election ile ya da basitçe `spring.batch.job.enabled=false` gibi birini kapatmakla). Veya bu işler tamamen queue bazlı ise problem yok. P2'de, *notification retry mekanizması* eğer @Scheduled ile bakıyorsa, cluster ortamında bir çözüm bulmalısınız. Basit çözüm: DB'ye kilit koymak (UPDATE with condition, vs.) ya da ShedLock kütüphanesini kullanmak (distributed lock for scheduled tasks). Bu ayrıntılar küçük ama unutulmamalı.

- **Scaling Limiters:** Uygulama kodunda belki sabit thread havuzu boyutları var (notificationExecutor vs.). Bunlar çok instance ile belki küçültülebilir. Yada global throughput limit vs. konabilir. Bu artık kapasite planlamaya giriyor.

- **Ek Güvenlik Sertleştirmeleri:** P2'de, prod ortamda dikkat edilecek bazı ekler:

- Uygulama bazlı CSRF, XSS vs. web konularımız yok pek (saf API).
- HTTP -> HTTPS yönlendirme (muhtemelen API gateway veya ingress halleder, ama Spring Boot'ta da redirect edilebilir).
- Güvenlik header'ları (Strict-Transport-Security, X-Content-Type-Options etc.) eklenebilir.

- **Audit Logging:** Belki bir DB tablosu ile kim ne zaman login oldu, kaç başarısız login denemesi oldu tutulabilir. Bu güvenlik izleme için önemli. Basitçe Spring Security Events yakalanarak yapılabilir. P2'de olması güzel ama olmazsa eksik sayılmaz.
- **DDoS koruması:** Rate limiting yaptık, ama ağır bir saldırı olursa belki WAF gibi şeylere ihtiyaç olur. Bu, ops tarafı aslında.

- **Data Encryption:** Kullanıcı şifreleri zaten hash'li. Başka hassas veri var mı? Belki yok ama ileride API key olursa en azından hmac ile tutulmalı vs. Bunlar not.

- **Monitoring & Alerting Ops:** Prod ortamda sadece uygulamanın kendini monitör etmesi yetmez, dışardan da monitör edilmeli. Bu devops tarafına giriyor (Prometheus, Grafana alarmlar vs.). Ancak bizim uygulama bazlı yapmamız gereken, olası hataları düzgün fırlatmak ki Prometheus'ta mesela *error rate* metriği hesaplanabilsin. Spring Boot Actuator'ın `/actuator/prometheus` metriğinde http server request metrics var, 5xx error count vs. geliyor. Onları Grafana alarmına bağlayabilirler. Bizim ekleyeceğimiz özel metriklerden belki "alertRuleEngine.failedMatches" falan yok ama gerek de yok. Yine de belki kendi custom metriğimizden alarm türetebiliriz (örn. chainhookProcessing.time ortalaması yükseldiyse alarm vs.). Bu artık SRE işi. Bizim kısım, metric doğru hesaplasın, log doğru tutsun.

- **İleri Yapısal İyileştirmeler:** Son olarak, mimari bir vizyon olarak P2'de belki şunlara karar verilebilir:

- İleride sistemi microservice'e bölecek miyiz? (Örneğin ingest ve processing ayrı olabilir). Şimdilik hayır belki. Ama büyürse seçenek.
- Event Sourcing'e geçer miyiz? Yani her chainhook event'ini bir event store'a (Kafka veya event log tablo) yazıp oradan projection yapma. Dokümanda bu önerilmişti, ama MVP için değil, belki v2 için [158] [96]. Not olarak kenarda durmalı.
- **CQRS ve Read Model:** Yine dokümanda bahsedilmiş, raporlama ve dashboard için ayrı bir denormalize tablo veya materialized view kullanılabilir [159] [160]. P2'de belki basit bir materialized view yapıp (örnekte alert_dashboard_mv) bazı istatistikleri denormalize tutabilirsiniz. Bu performansı uçurur raporlama için ama yazma tarafına yük katar (her insert sonrası view refresh gerekecek, ki concurrent refresh zordur). Timescale kullanılırsa belki continuous aggregate ile olabilir. Bu çok opsiyonel, belki ihtiyaç bile yok şu an. Yine de, eğer müşteri "toplam kaç uyarı üretildi" gibi sorgulara anlık cevap istiyorsa, MV mantıklı.

- **Makine Öğrenmesi vs.** Projenin geleceğinde anomali tespiti filan konuşulmuş ama bunlar MVP sonrası lüks özellikler. Mimarimiz esnek olduğundan, belki event geldiğinde ileride bir "AnomalyDetectorService" subscriber ekleriz vs. Şimdilik sadece not olarak kalır.

- **Son Kontroller ve Kullanıma Alma:** P2 sonunda, gerçek ortama çıkmadan bir **penetration test** ve **yük testi** faydalı olabilir. Pentest ile güvenlik açıkları aranır (OWASP top 10 vs.). Yük testinde de, örneğin 1 saatte 100k event gönderip sistem yetişebiliyor mu (Prometheus metrics incelenir, CPU, memory, DB IO vs. bakılır). Bu testlerin sonucuna göre belki minik ayarlar (thread pool büyütme, connection pool büyütme, ek indeks vs.) yapılabilir.

- **Claude Code ile Devam:** Claude Code'a burada belki daha az iş düşecek çünkü çoğu işi yapmış olacak. Ancak P2'de belki *"Write a Dockerfile for the Spring Boot app"*, *"Implement a RedisCacheConfig class for Spring Cache"*, *"Write a Kubernetes deployment YAML"* gibi istekler olabilir (Kod yazmasa da config yazar). Bu şekilde, insan hatasını azaltmak için bu son adımlarda da yardımı kullanabilirsiniz.

Bu yol haritasını özetlemek gerekirse:

- **P0 (Çekirdek + Güvenlik):** Domain modelleri, temel veri akışı (webhook ingest → db → kural kontrol → bildirim) uçtan uca ayağa kaldırılıyor. JWT ile auth ve HMAC ile webhook güvenliği ekleniyor. Kısacası "sistemin çalışır ilk versiyonu" çıkıyor.
- **P1 (İşlev Tamamlama + Optimizasyon):** Kalan önemli fonksiyonlar tamamlanıyor ve performans/güvenilirlik arttırılıyor. Kural motoru hızlandırılıyor (cache/index), bildirim mekanizması sağlamlaştırılıyor (retry/dlq), veri modeli ve sorgular optimize ediliyor (inheritance, indeks, vs.), gözlemlenebilirlik ekleniyor (metrikler, logging).
- **P2 (Prod Hazırlık + İleri İyileştirmeler):** Uygulama konteynerize ediliyor, configler prod seviyede yönetiliyor, yatay ölçek için çözümler ekleniyor (dağıtık cache, shedlock vs.), tüm güvenlik kontrolleri elden geçiriliyor, monitoring/alerting sistemi devreye alınıyor. Artık sistem gerçek kullanıma güvenle sunulabilir hale geliyor.

Her bir faz sonunda, o aşamaya ait **kodların derlenip çalıştığı**, test senaryolarını geçtiği ve belgelendiği kontrol edilmelidir. Bu iteratif yaklaşım sayesinde, en kritik ihtiyaçları önce karşılayıp, adım adım iyileştirmeler yaparak hedeflenen enterprise-grade mimariye ulaşacağız.

Sonuç: Bu kapsamlı analiz ve plan doğrultusunda, Stacks Blockchain Smart Contract Monitoring sisteminin mevcut güçlü yönlerini (Clean Architecture, zengin domain modeli, esnek kural yapısı) koruyarak; eksik yönlerini de endüstri standardı çözümlerle iyileştirerek **production-ready** hale getirebiliriz. Önerilen mimari düzenlemeler (örneğin event'lerin ve kuralların polimorfik tasarımı, event-driven pattern'lerin kullanımı) sistemi uzun vadede **ölçeklenebilir** ve **bakımı kolay** kılacaktır. Güvenlik katmanı (JWT, HMAC, rate-limit) eklendiğinde ve performans darboğazları giderildiğinde, sistem kurumsal ortamlarda güvenle kullanılabilecek bir MVP olacaktır. Sonraki aşamalarda (P2 sonrası) gerçek zamanlı izleme ve belki ileri seviye analizler (anomaly detection, ML integration) eklemek de mümkün olacak, ancak önce sağlam bir çekirdek kurmak esastır. Bu plan dahilinde Claude Code'dan da parça parça yardım alarak geliştirmeyi hızlı ve hatasız şekilde ilerletebilir, her fazın sonunda çalışan bir ürün elde edebiliriz.

Kaynaklar:

- Proje teknik dökümanları ve kod incelemeleri 93 68
- Endüstri en iyi uygulamalarından çıkarılan öneriler 23 116
- Spring Boot ve ilgili ekosistem dokümantasyonları (Security, Cache, Batch, vs.)
- Blockchain monitoring sistemleri üzerine araştırma notları (anomaly detection, event streaming gibi ileri konular).

---

1 2 3 4 8 9 16 21 24 26 27 28 29 30 31 32 34 35 36 39 40 41 42 43 58 63 64 65 66 67 68 69 78 90 93 94 95 Stacks Chain Monitor Backend – Sistem Yapısı ve Kod Temelleri.pdf
file://file_000000003540720eb2442333f1f2ae9d

5 6 7 12 13 17 18 19 20 33 37 38 48 49 50 51 52 55 56 57 59 60 61 62 72 73 75 76 77 79 83 120 128 129 150 151 152 154 155 156 157 stacks-backend-tam-dokumantasyon.md
file://file_00000000288471f5a648bc85de92252a

10 11 14 15 22 23 25 44 45 46 47 53 54 70 71 74 80 81 82 84 85 86 87 88 89 91 92 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 121 122 123 124 125 126 127 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 153 158 159 160

Stacks Blockchain Smart Contract Monitoring System.pdf

file://file_000000004be071f5a6e0657cc989aa19