Технически Университет - София

Разпределена мулти-тенант платформа за управление на конфигурации и политики с версониране

курсова работа по

„Нерелационни бази данни“

Имена на студент: Алекс Цветанов

Факултетен номер: 121222225

Група: 41

Специалност: **Компютърно и софтуерно инженерство**

Образователно-квалификационна степен: **бакалавър**

София, 13 февруари 2026 г.

СЪДЪРЖАНИЕ

РЕЗЮМЕ

Настоящата курсова работа разглежда проектирането и реализацията на система за разпределено управление на конфигурации и политики с поддръжка на версиониране. Съвременните софтуерни системи, изградени по микросървисна архитектура и разположени в облачни среди, изискват способност за централизирано управление на конфигурациите с възможност за проследяване на промените, връщане към предишни версии и поддържка на множество среди.

Основната цел на проекта е създаването на производствена платформа, която да осигурява неизменяемо версиониране на JSON конфигурации с пълна поддръжка на branching, rollback и diff операции. Системата трябва да осигурява мулти-тенант изолация на данните и пълен одитен запис на всички операции.

Теоретичната обосновка на избора на технологии се базира на анализ на специфичните изисквания към системите за управление на конфигурации. Документно-ориентираните бази данни се оказват архитектурно подходящи за този домейн поради естественото съвпадение между JSON конфигурациите и документния модел. MongoDB е избрана като конкретна реализация, предоставяща необходимата производителност, гъвкавост на схемата и възможности за хоризонтално мащабиране.

Практическата реализация включва Node.js бекенд с Express.js уеб фреймуърк и MongoDB база данни. Системата реализира модел на неизменяеми версии, при който всяка версия на конфигурация се съхранява като отделен документ с SHA256 контролна сума. Поддържат се операции за създаване на версии, връщане към предишни състояния, разклоняване на конфигурации и сравнение между версии.

Ключови функционалности на системата включват управление на конфигурации с операции за създаване, четене, актуализиране и изтриване; версиониране с пълна история на промените и възможност за възстановяване на произволна версия; разклоняване за паралелна работа върху различни варианти на конфигурации; разлики между версии с детайлно сравнение на структурните промени; мулти-тенант архитектура с пълна изолация на данните между клиенти; одитен запис с регистриране на всички операции за съответствие.

Резултатите от разработката демонстрират, че използването на

документно-ориентирана база данни за версиониране на конфигурации предоставя значителни предимства спрямо релационния подход, включително по-проста архитектура, по-добра производителност при четене и естествена поддръжка на йерархични структури.

Ключови думи: NoSQL, MongoDB, версиониране, конфигурации, мулти-тенант, JSON, Node.js, документно-ориентирани бази данни, разпределени системи.

I. ВЪВЕДЕНИЕ

1.1. Актуалност на проблема

В съвременните разпределени софтуерни системи управлението на конфигурации представлява критично предизвикателство [3, 15]. Микросървисната архитектура, облачните инфраструктури и динамичните runtime политики изискват способност за бързо променяне на конфигурацията без прекъсване на услугите. Традиционните подходи за съхранение на конфигурации чрез файлови системи, релационни бази данни или key-value хранилища страдат от фундаментални ограничения при опит за реализиране на пълноценно версиониране.

Конкретният бизнес проблем, който настоящият проект адресира, произтича от няколко често срещани сценария в съвременната софтуерна разработка, включително управление на feature flags [10]. **Feature Flags** представляват механизъм, при който разработчици трябва да активират или деактивират функционалности в production среда без необходимост от redeploy на приложението. Това изисква възможност за моментално прилагане на конфигурационни промени с пълен контрол върху състоянието. **A/B Testing** налага поддръжка на различни конфигурации за различни потребителски сегменти, което изисква способност за управление на множество версии на една и съща конфигурация, действащи паралелно. **Infrastructure as Code** подходът изисква версиониране на инфраструктурни дефиниции с възможност за връщане към стабилни състояния при проблеми. **Policy Engines** изискват динамични правила за достъп и сигурност, които могат да се променят в реално време с възможност за одит на промените. **Multi-tenant SaaS** архитектурата изисква изолирани конфигурации за всеки клиент с пълна логическа и физическа сепарация на данните.

При всички тези сценарии се налагат общи изисквания към системата за управление на конфигурации: възможност за моментален rollback при откриване на проблем, поддръжка на пълна история на промените с информация за автор и времева марка, възможност за сравнение между различни версии с детайлно представяне на разликите, както и възможност за експериментиране чрез разклоняване на конфигурациите в отделни branch-ове.

1.2. Цели и задачи на проекта

Основната цел на разработената платформа е създаването на производствена система за управление на конфигурации, която осигурява неизменяемо версиониране на JSON конфигурации с пълна поддръжка на разклоняване, връщане към предишни състояния и операции за сравнение между версии.

Платформата трябва да осигурява **неизменяемо версиониране**, при което всяка версия на конфигурация се съхранява като неизменяем документ с криптографска контролна сума, гарантираща целостта на данните. Системата трябва да поддържа **разклоняване и връщане към предишни състояния**, позволявайки създаването на експериментални клонове за тестване на нови функционалности и моментално връщане към стабилни версии при необходимост. Необходима е **пълна логическа изолация на данни** между различни организации чрез мулти-тенант архитектура, осигуряваща сигурност и конфиденциалност. Системата трябва да поддържа **одитен запис** на всички операции с конфигурации, регистрирайки автор, времева марка и тип на операцията за целите на съответствие с регулаторни изисквания. Необходимо е **автоматично откриване на разлики** между версии с детайлно сравнение на структурните промени в JSON документите. Платформата трябва да осигурява **поддръжка на множество среди**, позволявайки едновременно управление на development, staging и production конфигурации.

1.3. Обосновка на избора на нерелационна база данни

Изборът на документно-ориентирана NoSQL база данни, конкретно MongoDB, се базира на детайлен анализ на специфичните изисквания към системите за управление на конфигурации [13, 18]. Традиционните релационни бази данни се оказват неподходящи за този домейн поради фундаментални архитектурни различия в моделите на данни.

Полуструктурираният характер на данните представлява основно предизвикателство за релационния модел. Конфигурациите са естествено JSON документи с произволна дълбочина и йерархия, като различни конфигурации могат да имат коренно различна структура. Релационният модел изисква или използване на EAV анти-патерн, или чести schema миграции при промяна на структурата, които водят до

технически дълг и сложна поддръжка.

Гъвкавостта на схемата при документно-ориентираните бази позволява съхраняване на различни конфигурации с различна структура без нужда от промяна на database schema. Това е особено важно за системи с динамични конфигурации, където структурата може да се променя често без предизвестие.

Версионирането чрез документи се реализира естествено при документния модел, където всяка версия е представена като отделен документ. Този подход елиминира необходимостта от сложни join операции и осигурява атомарен достъп до цялата конфигурация като единица, което е критично за производителността при четене.

Хоризонталното мащабиране е заложено в архитектурата на MongoDB чрез нативна поддръжка на sharding. Това позволява разпределение на данни по tenantId, което е критично за мулти-тенант системи с голям брой клиенти и високи изисквания към производителността.

Query операциите върху вложени структури се реализират ефективно чрез MongoDB агрегационния pipeline, който позволява сложни аналитични заявки върху вложени JSON полета без нужда от денормализация на данните. Това е съществено предимство при работа със сложни конфигурационни документи.

CAP теоремата определя компромисите между консистентност, наличност и толерантност към разделяния при разпределени системи. За система за конфигурации, където четенията са много по-чести от писанията, MongoDB предлага отличен баланс с възможност за настройка на consistency level според конкретните изисквания.

Проектът демонстрира, че за специфичния домейн на версиониране на конфигурации документно-ориентираните NoSQL бази данни не представляват компромис, а са архитектурно оптималното решение [9].

II. ТЕОРЕТИЧНА И ТЕХНОЛОГИЧНА ОБОСНОВКА

2.1. Нерелационни бази данни

2.1.1. Основни характеристики на NoSQL

Нерелационните бази данни, известни още като NoSQL, се отличават от релационния модел по няколко ключови характеристики, които ги правят подходящи за специфични сценарии на употреба [18].

Липсата на фиксирана схема представлява фундаментална разлика спрямо SQL базите. Докато в релационния модел схемата трябва да бъде дефинирана предварително и всички записи трябва да следват тази структура, NoSQL базите позволяват динамична структура на записите. Тази характеристика е особено ценна за системи с полуструктурирани данни, където различни записи могат да имат различни полета според специфичните нужди.

Разпределеното съхранение е заложено в архитектурата на NoSQL базите от самото начало. Вместо вертикално мащабиране чрез добавяне на ресурси към един сървър, тези бази са проектирани за хоризонтално мащабиране чрез разпределяне на данни между множество сървъри. Механизмът sharding позволява автоматично разпределение на данните според дефиниран ключ, осигурявайки висока наличност и отказоустойчивост [12].

BASE моделът заменя традиционния ACID модел при много NoSQL системи [4]. Това е съзнателен компромис, който жертва незабавната консистентност в полза на по-висока наличност при разпределени среди. BASE означава Basically Available, Soft state, Eventually consistent и е особено подходящ за приложения, където краткотрайната несъвместимост между реплики е приемлива в името на по-добрата производителност и наличност.

2.1.2. Типове нерелационни бази данни

Съществуват четири основни категории нерелационни бази данни, всяка със специфични силни страни и подходящи сценарии на употреба. Таблица 2.1 представя сравнителен преглед на категориите.

Таблица 1. Категории NoSQL бази данни

Тип	Представители	Подходящи сценарии
Документно-ориентирани	MongoDB, CouchDB, RavenDB	JSON документи, йерархични данни
Key-Value	Redis, DynamoDB, Riak	Кеширане, сесии, бърз достъп
Колонно-ориентирани	Cassandra, HBase, Bigtable	Time-series, аналитика
Графови	Neo4j, ArangoDB, OrientDB	Релации, препоръчителни системи

Документно-ориентираните бази съхраняват данни като JSON или BSON документи с вложена структура, което ги прави идеални за полуструктурирани данни. Key-Value хранилищата предоставят най-простия модел за бърз достъп по ключ, често използван за кеширане. Колонно-ориентираните бази оптимизират записа и четенето на големи обеми данни чрез съхраняване по колони вместо по редове. Графовите бази са специализирани за данни със сложни връзки между обекти.

2.2. Избор на MongoDB

За настоящия проект е избрана MongoDB като конкретна реализация на документно-ориентирана база данни. MongoDB съхранява данни в BSON формат, който представлява бинарна сериализация на JSON с допълнителна поддръжка за типове данни като Date и Binary [3, 13].

2.2.1. Защо документно-ориентиран модел

Конфигурационните данни са естествено документи с характеристики, които съвпадат идеално с документно-ориентирания модел. Конфигурациите притежават **йерархична структура** с вложени обекти, като например полета от типа `retryPolicy.maxRetries`, които представляват дълбоко вложени конфигурационни параметри. Те са **полуструктурирани**, което означава, че различни конфигурации могат да имат различни полета според специфичните си нужди. Изискват **атомарен достъп**, тъй като цялата конфигурация се чете и записва като единица. Всяка версия е **самостоятелен документ**, което позволява независимо управление и версиониране [14].

2.2.2. CAP теорема и MongoDB

CAP теоремата дефинира трите основни свойства на разпределените системи: консистентност, наличност и толерантност към разделяния [4]. MongoDB позволява настройка на тези компромиси чрез параметрите write concern и read preference, давайки възможност за балансиране според конкретните изисквания.

Консистентността е конфигурируема, като write majority осигурява силна консистентност при запис, докато други настройки позволяват по-висока производителност. **Наличността** е висока чрез механизма replica sets, който осигурява автоматично възстановяване при отказ на сървър. **Толерантността към разделяния** се реализира чрез нативната поддръжка на sharding, която позволява разпределение на данните между множество сървъри.

За система за конфигурации, където четенията доминират над писанията, е оптимален балансът с eventual consistency за четения и strong consistency за критични операции по запис.

2.3. Технологичен стек

2.3.1. Backend платформа

Node.js е избран като сървърна платформа поради няколко ключови характеристики, които го правят идеален за интеграция с MongoDB [17]. Платформата осигурява нативна JSON поддръжка, която съвпада идеално с BSON формата на MongoDB и позволява директна сериализация и десериализация на данни. Async I/O моделът е ефективен при операции, обвързани с вход и изход, като заявки към база данни, тъй като позволява на една нишка да обслужва множество едновременни връзки. Използването на единен език JavaScript в целия стек опростява разработката и поддръжката. Богатата екосистема от библиотеки включва Mongoose ODM, който осигурява високо ниво на абстракция при работа с MongoDB.

Express.js предоставя минималистичен, но мощен уеб фреймуърк [16]. Архитектурата му е базирана на middleware компоненти за cross-cutting concerns като логиране и аутентикация, които се изпълняват последователно при обработка на заявки. Router системата позволява структуриране на RESTful API с ясно разделение на

маршрутите. Лесната интеграция с библиотеки за валидация и аутентикация опростява добавянето на допълнителна функционалност.

2.3.2. Mongoose ODM

Mongoose Object Document Mapper предоставя високо ниво на абстракция при работа с MongoDB [1, 2]. Основните функционалности включват дефиниране на схема с валидация и автоматично преобразуване на типове, което осигурява цялостност на данните преди запис в базата. Middleware hooks позволяват изпълнение на код преди и след операции, като например изчисляване на контролна сума преди запис. Query building с метод chaining позволява конструиране на сложни заявки чрез свързване на методи. Population механизмът управлява релациите между документи и автоматично зареждане на свързани данни.

2.3.3. Поддържащи технологии

Освен основния технологичен стек, системата интегрира допълнителни технологии за специфични нужди, включително Docker за контейнеризация [6]. Таблица 2.2 представя преглед на тези технологии.

Таблица 2. Поддържащи технологии

Технология	**Предназначение**
Docker	Контейнеризация за преносимо разгръщане
JWT	Stateless аутентикация между микросървиси
Helmet.js	Security headers за защита от чести уеб атаки
Joi	Схемна валидация за API входни данни
Winston	Структурирано логиране с множество транспорти
Rate-limiter-flexible	Защита срещу злоупотреба и DDoS атаки

2.4. Сравнение с релационен подход

2.4.1. Проблеми при релационния модел

Релационният подход за версиониране на конфигурации води до няколко значителни проблеми, които го правят неподходящ за този домейн.

Entity-Attribute-Value анти-патернът е често използван подход за съхраняване на полуструктурирани данни в релационни бази. Този модел обаче страда от сериозни

недостатъци: загуба на type safety поради съхраняване на всички стойности като низове, необходимост от сложни self-join операции за реконструкция на целия обект, и лоша производителност при голяма дълбочина на вложеност.

Версионирането при релационни бази изисква компромиси между различни подходи, всеки със своите недостатъци. Копирането на цели редове при всяка нова версия води до излишно увеличаване на обема данни. Delta таблиците, съхраняващи само разликите, са сложни за заявки и изискват сложна логика за възстановяване на пълното състояние. JSON колоните предлагат частично решение, но с ограничена индексация и вторична поддръжка спрямо документно-ориентираните бази.

Схемната ригидност на релационния модел изисква миграции при промяна на структурата на конфигурацията, което е неприемливо за динамични системи с често променящи се изисквания. При документно-ориентираните бази промените в структурата се обработват автоматично без нужда от явни миграции.

Join overhead при възстановяване на пълна конфигурация с одитен запис изисква множество join операции между конфигурационни, версионни и одитни таблици, което деградира производителността при четене.

2.4.2. Performance сравнение

Таблица 2.3 представя сравнителен анализ на производителността между MongoDB и PostgreSQL с JSONB колони за конфигурационни данни.

Таблица 3. Сравнение MongoDB vs PostgreSQL за конфигурационни данни

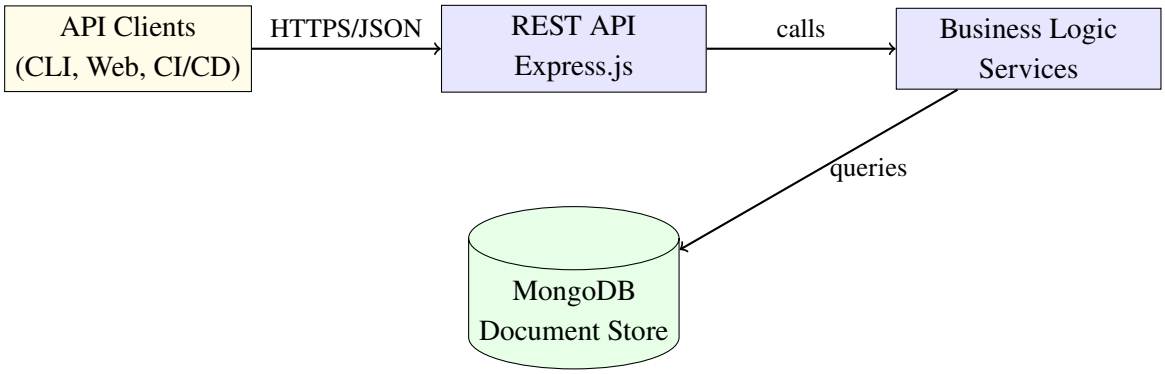
Операция	**MongoDB**	**PostgreSQL (JSONB)**
Read версия (indexed)	O(1) документ	O(1) ред + JSON parse
Write нова версия	O(1) insert	O(1) insert
Diff две версии	2 документа	2 реда + JSON parse
Query вложено поле	Native dot notation	JSONB operators
Horizontal scaling	Native sharding	Външни решения

Сравнението показва, че MongoDB предоставя по-добра производителност за специфичните операции, характерни за системи за версииране на конфигурации, особено при мащабиране и работа с вложени структури.

III. АРХИТЕКТУРА И РЕАЛИЗАЦИЯ

3.1. Обща архитектура

Системата следва многослойна архитектура с ясно разделение на отговорностите между отделните компоненти [11]. Фигура 1 представлява архитектурната диаграма на системата, илюстрираща основните компоненти и връзките между тях.



Фигура 1. Архитектурна диаграмата на системата

Архитектурата се състои от три основни слоя. Първият слой е REST API слоят, изграден с Express.js, който обработва HTTP заявки от клиенти и осигурява маршрутизация, валидация и аутентикация. Вторият слой е бизнес логиката, реализирана чрез сервизни класове, които съдържат основната функционалност за управление на конфигурации и версониране. Третият слой е слоят за достъп до данни, който използва MongoDB за съхранение и извличане на документи.

3.2. Моделиране на данните

Системата използва четири основни колекции за съхранение на данни, всяка със специфично предназначение и структура, оптимизирана за конкретни операции.

3.2.1. Колекция tenants

Колекцията tenants съхранява информация за организациите, използващи системата, и осигурява изолация на данните между различни клиенти. Необходима е **пълна логическа изолация на данни** между различни организации чрез мулти-тенант архитектура, осигуряваща сигурност и конфиденциалност [5]. Листинг 3.1 представя

примерна структура на документ от тази колекция.

```
{
    "_id": "tenantA",
    "name": "Acme Corp",
    "description": "Enterprise customer",
    "settings": {
        "maxConfigs": 100,
        "maxVersionsPerConfig": 1000,
        "retentionDays": 365
    },
    "isActive": true,
    "createdAt": ISODate("2026-02-01T00:00:00Z"),
    "updatedAt": ISODate("2026-02-01T00:00:00Z")
}
```

Listing 1: Структура на Tenant документ

Индексът `isActive: 1` осигурява бързо филтриране на активни tenants при заявки за административни операции.

3.2.2. Колекция configs

Колекцията configs съхранява метаданни за конфигурациите и указатели към активните версии за всяка среда. Това е централната колекция за управление на конфигурации, която поддържа информация за имената, средите, таговете и архивния статус. Листинг 3.2 представя примерна структура.

```
{
    "_id": ObjectId("65b8a..."),
    "tenantId": "tenantA",
    "name": "payment-service",
    "description": "Payment gateway configuration",
    "environments": ["dev", "staging", "prod"],
    "activeVersions": {
        "dev": 12,
        "staging": 10,
        "prod": 8
    },
    "tags": ["critical", "payments"],
    "isArchived": false,
    "createdBy": "admin@acme.com",
    "createdAt": ISODate("2026-02-01T10:00:00Z"),
    "updatedAt": ISODate("2026-02-13T15:30:00Z")
```

```
}
```

Listing 2: Структура на Config документ

Колекцията configs използва няколко индекса за оптимизация на заявките. Съставният индекс `tenantId: 1, name: 1` осигурява уникалност на имената на конфигурациите в рамките на един tenant. Индексът `tenantId: 1, isArchived: 1` ускорява заявките за списъци с филтриране по архивен статус. Индексът `tags: 1` поддържа филтриране по тагове.

3.2.3. Колекция config_versions

Колекцията config_versions представлява сърцето на системата за версиониране, съхранявайки неизменяеми версии на конфигурации с криптографски контролни суми. Всяка версия е самостоятелен документ, който съдържа пълните данни на конфигурацията за конкретна версия. Листинг 3.3 представя структурата на документ.

```
{
    "_id": ObjectId("65b8b..."),
    "configId": ObjectId("65b8a..."),
    "tenantId": "tenantA",
    "version": 12,
    "branch": "main",
    "parentVersion": 11,
    "data": {
        "retryPolicy": {
            "maxRetries": 5,
            "backoff": "exponential",
            "initialDelay": 1000
        },
        "featureFlags": {
            "newCheckout": true,
            "betaPayments": false,
            "fraudDetection": true
        },
        "rateLimits": {
            "perMinute": 1200,
            "burst": 200
        },
        "providers": ["stripe", "paypal"]
    },
```

```
    "checksum": "a3f7c8...e2d9b1",
    "changeLog": "Increased retry limit and added fraud detection",
    "createdBy": "admin@acme.com",
    "createdAt": ISODate("2026-02-13T15:30:00Z"),
    "isArchived": false
}
```

Listing 3: Структура на ConfigVersion документ

Критичните индекси за тази колекция са дефинирани както следва. Листинг 3.4 представя дефинициите на индексите.

```
db.config_versions.createIndex({ configId: 1, version: -1 })
db.config_versions.createIndex({ tenantId: 1, branch: 1 })
db.config_versions.createIndex({ createdAt: -1 })
db.config_versions.createIndex({ checksum: 1 })
```

Listing 4: Дефиниции на индекси за config_versions

Съставният индекс `configId: 1, version: -1` е от особено значение, тъй като осигурява константна сложност при достъп до конкретна версия и ефективно сортиране по версия в низходящ ред. Индексът по контролна сума позволява бърза проверка за съществуващи версии с идентично съдържание.

3.2.4. Колекция audit_logs

Колекцията audit_logs поддържа непроменяем запис на всички операции, извършвани върху конфигурациите. Този одитен запис е критичен за съответствие с регулаторни изисквания и за проследяване на промените. Листинг 3.5 представя структурата.

```
{
    "_id": ObjectId("65b8c..."),
    "entityType": "config",
    "entityId": "65b8a...",
    "tenantId": "tenantA",
    "action": "ROLLBACK",
    "performedBy": "admin@acme.com",
    "performedAt": ISODate("2026-02-13T16:00:00Z"),
    "metadata": {
        "environment": "prod",
```

```
        "fromVersion": 12,
        "toVersion": 10,
        "reason": "Fraud detection causing latency"
    },
    "ipAddress": "192.168.1.100",
    "userAgent": "Mozilla/5.0..."
}
```

Listing 5: Структура на AuditLog документ

Индексите за одитни заявки са дефинирани както следва в листинг 3.6.

```
db.audit_logs.createIndex({ entityId: 1, performedAt: -1 })
db.audit_logs.createIndex({ tenantId: 1, performedAt: -1 })
db.audit_logs.createIndex({ action: 1, performedAt: -1 })
db.audit_logs.createIndex({ performedBy: 1, performedAt: -1 })
```

Listing 6: Индекси за audit_logs

3.3. Основни операции

3.3.1. Създаване на нова версия

Алгоритъмът за създаване на версия следва последователност от стъпки, които осигуряват цялостност и проследяемост на данните. Процесът започва с валидация на входните данни чрез Joi схема, която проверява структурата и типовете на JSON конфигурацията. Следваща стъпка е автентикация и авторизация чрез JWT токени и проверка на RBAC роли. Системата извлича текущата версия за полето parentVersion, което създава връзка във версионната история. След това се изчислява SHA256 контролната сума на данните за гарантиране на неизменяемост. Автоматично се генерира changeLog чрез сравнение с предишната версия. Новият документ се записва атомарно в базата данни и се създава одитен запис. Листинг 3.7 представя имплементацията на сервиза за създаване на версии.

```
class VersioningService {
    static async createVersion({
        configId, tenantId, data, branch,
        parentVersion, changeLog, createdBy
```

```
    }) {
        const nextVersion = await ConfigVersion
            .getNextVersion(configId);

        const version = new ConfigVersion({
            configId,
            tenantId,
            version: nextVersion,
            branch,
            parentVersion: parentVersion ||
                await this.getCurrentVersion(configId, branch),
            data,
            changeLog: changeLog ||
                await this.generateChangeLog(configId, data),
            createdBy
        });

        await version.save();
        await AuditService.log({ ... });

        return version;
    }
}
```

Listing 7: Имплементация на създаване на версия

3.3.2. Операция за връщане към предишна версия

Операцията rollback променя само указателя в полето activeVersions, без да изтрива данни, което осигурява моментално връщане към стабилно състояние. Този подход запазва пълната история на версиите и позволява бъдещи връщания към по-нови версии при необходимост. Листинг 3.8 представя имплементацията.

```
static async rollback({
    configId, tenantId, environment,
    targetVersion, performedBy, reason
}) {
    const config = await Config.findOne({
        _id: configId, tenantId
    });

    const currentVersion = config.activeVersions[environment];
```

```
    config.activeVersions[environment] = targetVersion;
    await config.save();

    await AuditService.log({
        entityType: 'config',
        action: 'ROLLBACK',
        metadata: {
            environment,
            fromVersion: currentVersion,
            toVersion: targetVersion,
            reason
        }
    });
}
```

Listing 8: Имплементация на rollback операция

Времевата сложност на операцията е константна, тъй като включва единствено актуализация на config документ с промяна на едно поле във вграден обект.

3.3.3. Diff алгоритъм и сравнение на версии

Diff алгоритъмът използва библиотеката deep-diff за структурно сравнение на JSON обекти. Алгоритъмът обхожда дълбоко вложените структури и идентифицира добавени, изтрити и модифицирани полета. Листинг 3.9 представя имплементацията на Diff сервиза.

```
class DiffService {
    static computeDiff(oldData, newData) {
        const differences = deepDiff.diff(oldData, newData);

        return {
            hasChanges: !!differences,
            changes: differences?.map(diff => ({
                type: this.getChangeType(diff),
                path: diff.path?.join('.'),
                oldValue: diff.lhs,
                newValue: diff.rhs
            })) || [],
            summary: this.generateSummary(differences)
        };
    }

    static getChangeType(diff) {
```

```
        if (diff.kind === 'N') return 'added';
        if (diff.kind === 'D') return 'deleted';
        if (diff.kind === 'E') return 'modified';
        return 'unknown';
    }
}
```

Listing 9: Имплементация на diff алгоритъм

3.4. Примери за агрегационни заявки

MongoDB агрегационният pipeline предоставя мощни възможности за анализ на данните. Листинг 3.10 представя заявка за статистика по месеци, която групира версиите по година и месец на създаване.

```
db.config_versions.aggregate([
    { $match: { tenantId: "tenantA" } },
    {
        $group: {
            _id: {
                year: { $year: "$createdAt" },
                month: { $month: "$createdAt" }
            },
            count: { $sum: 1 }
        }
    },
    { $sort: { "_id.year": -1, "_id.month": -1 } }
])
```

Listing 10: Статистика по месеци

Листинг 3.11 представя заявка за извличане на последната версия за всяка конфигурация чрез групиране и избор на първи елемент след сортиране.

```
db.config_versions.aggregate([
    { $match: { tenantId: "tenantA" } },
    { $sort: { configId: 1, version: -1 } },
    {
        $group: {
            _id: "$configId",
            latestVersion: { $first: "$version" },
```

```
            data: { $first: "$data" },
            createdAt: { $first: "$createdAt" }
        }
    }
])
```

Listing 11: Заявка за последни версии

3.5. Стратегия за шардиране

За production deployment с множество tenants се препоръчва използване на shard key по tenantId. Този подход осигурява изолация на данните, като всички данни на даден tenant се съхраняват на един shard, което подобрява сигурността и опростява backup операциите. Разпределението на различни tenants по различни shards осигурява балансирано натоварване. Заявките, филтриращи по tenantId, попадат на един shard, което елиминира необходимостта от scatter-gather операции. Листинг 3.12 представя конфигурацията за шардиране.

```
sh.enableSharding("config_versioning")
sh.shardCollection(
    "config_versioning.config_versions",
    { tenantId: 1, configId: 1 }
)
```

Listing 12: Конфигурация за шардиране

3.6. Имплементация на сигурност

3.6.1. Контрол на достъп базиран на роли

Системата използва модел за контрол на достъп базиран на роли (RBAC), който дефинира три основни роли с различни нива на привилегии. Таблица 3.1 представя ролевия модел.

Таблица 4. Ролева модел за контрол на достъп

Роля	Tenant	Config	Version
admin	CRUD	CRUD + Archive	CRUD + Rollback
editor	Read	CRUD	CRUD
viewer	Read	Read	Read

Ролята admin притежава пълни права за управление на tenants, конфигурации и версии, включително архивиране и rollback операции. Ролята editor може да създава и редактира конфигурации и версии, но няма права за административни операции върху tenants или rollback. Ролята viewer има само права за четене на всички ресурси.

3.6.2. Ограничаване на честотата на заявките

Защитата срещу злоупотреба се реализира чрез ограничаване на броя заявки от един IP адрес или tenant в определен времеви прозорец, като се използва JWT за автентикация [7]. Листинг 3.13 представя конфигурацията.

```
const limiter = rateLimit({
    windowMs: 15 * 60 * 1000,  // 15 минути
    max: 1000,                 // заявки на IP
    keyGenerator: (req) =>
        req.user?.tenantId || req.ip
});
```

Listing 13: Конфигурация за rate limiting

IV. ЗАКЛЮЧЕНИЕ

4.1. Резултати от проекта

Разработената система за разпределено управление на конфигурации и политики с поддръжка на версиониране успешно демонстрира приложението на документно-ориентирани нерелационни бази данни за решаване на реален индустриален проблем в областта на управлението на динамични конфигурации в разпределени системи.

Проектът постига пълноценна производствена система с изграден REST API, механизми за автентикация и авторизация, както и пълен одитен запис на операциите. Системата реализира неизменяемо версиониране с криптографски контролни суми чрез SHA256 алгоритъм и поддържа пълна история на промените за всяка конфигурация. Мулти-тенант архитектурата осигурява изолация на данните чрез стратегия за шардиране базирана на tenantId. Поддържат се разклоняване и връщане към предишни състояния, което дава възможност за експериментални конфигурации и моментално връщане към стабилни версии при проблеми. Реализиран е структурен diff алгоритъм за автоматично откриване на разлики между JSON версии. Системата поддържа едновременно управление на множество среди, включително development, staging и production. Оптимизирани индекси осигуряват ефективен достъп до активни версии, история и одитни записи.

4.2. Аргументация за избор на NoSQL база данни

Проектът предоставя убедително доказателство за превъзходството на документно-ориентирания подход за разглеждания домейн на управление на конфигурации.

Относно производителността, системата осигурява константна сложност при достъп до всяка версия чрез съставния индекс `configId: 1, version: -1`. При релационен модел същата операция изисква или каскадни join операции между множество таблици, или скъпо парсиране на JSONB колони, което деградира производителността.

Гъвкавостта на схемата е друго съществено предимство. Липсата на фиксирана схема позволява различни конфигурации да имат коренно различна структура без необходимост от schema миграции. Това е критично за динамични системи, където

структурата на конфигурациите се променя често.

Мащабируемостта се реализира чрез нативно хоризонтално мащабиране чрез шардиране по tenantId, което осигурява линейна мащабируемост с нарастване на броя клиенти. Този подход е значително по-ефективен от вертикалното мащабиране при релационни бази.

Заявките върху вложени структури се реализират ефективно чрез MongoDB агрегационния pipeline, който позволява аналитични заявки директно върху вложени JSON полета без нужда от денормализация или сложни join операции.

4.3. Практическа приложимост

Системата може директно да се прилага в редица индустриални сценарии. Платформите за управление на feature flags, като LaunchDarkly и Unleash, изискват сходна функционалност за версиониране и разклоняване на конфигурации. CI/CD pipeline системите се нуждаят от версиониране на конфигурации за различни среди. Policy-as-Code двигателите, като Open Policy Agent, изискват версиониране на политики с възможност за rollback. Микросървисните архитектури изискват централизирано управление на конфигурации с мулти-тенант поддръжка. Системите за A/B тестване изискват управление на различни конфигурации за различни потребителски сегменти.

4.4. Бъдещи подобрения

Въпреки че текущата реализация покрива всички основни изисквания, съществуват няколко направления за бъдещи подобрения.

Интеграцията с MongoDB Change Streams би позволила реализация на механизъм за push notifications при промяна на конфигурация, което би елиминирало необходимостта от polling от страна на клиентите.

Разширяването на функционалността за разклоняване с три-way merge алгоритъм би позволило автоматично разрешаване на конфликтни ситуации при сливане на разклонения, подобно на функционалността в системите за контрол на версии като Git.

Добавянето на JSON Schema validation на ниво MongoDB би осигурило по-строг контрол върху структурата на конфигурационните данни [8], което би повишило

цялостността и надеждността на системата.

Client-side field-level encryption би осигурило допълнителна сигурност за чувствителни конфигурации, като криптографски ключове или пароли, съхранявани в системата.

Интеграцията с Redis като кеширащ слой би позволила кеширане на активни конфигурации с време за достъп под милисекунда, което би подобрило производителността при високо натоварване.

4.5. Заключителни бележки

Проектът демонстрира, че изборът на подходяща база данни е критично архитектурно решение, което влияе върху производителността, мащабируемостта и поддръжката на системата. За домейни с полуструктурирани данни, йерархични структури и нужда от версиониране, документно-ориентираните нерелационни бази данни не представляват компромис, а са архитектурно оптималното решение.

Реализираната платформа представлява пълноценно производствено решение, което може да бъде директно интегрирано в съвременни cloud-native системи за управление на конфигурации. Използването на MongoDB като документно-ориентирана база данни се оказа оправдан избор, осигуряващ необходимата гъвкавост, производителност и възможности за мащабиране.

ЛИТЕРАТУРА

[1] Automattic Inc. Mongoose documentation, 2024.

[2] Kyle Banker, Peter Bakkum, Shaun Verch, Doug Garrett, and David Hows. *MongoDB in Action*. Manning Publications, 2011.

[3] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, 3rd edition, 2019.

[4] Eric Brewer. Cap twelve years later: How the r̈ulesḧave changed, 2012.

[5] Fred Chong, Gianpaolo Carraro, and Roger Wolter. Multi-tenant data architecture. *Microsoft Developer Network*, 2006.

[6] Docker Inc. Docker documentation, 2024.

[7] Michael B. Jones, John Bradley, and Nat Sakimura. Json web token (jwt) rfc 7519, 2015.

[8] JSON Schema Org. Json schema validation specification, 2024.

[9] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[10] LaunchDarkly. Feature flagging guide, 2024.

[11] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[12] Microsoft Azure. Data partitioning guidance, 2024.

[13] MongoDB Inc. Mongodb documentation, 2024.

[14] MongoDB Inc. Mongodb indexing strategies, 2024.

[15] Sam Newman. Building microservices. *O'Reilly Media*, 2015.

[16] OpenJS Foundation. Express.js documentation, 2024.

[17] OpenJS Foundation. Node.js documentation, 2024.

[18] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.