**Software Requirements Specification (SRS)**

**Проект: Система мониторинга IoT-устройств**

## **1. Цели проекта (Purpose)**

Создать систему мониторинга данных от IoT-устройств, которая обеспечивает:

* Сбор данных в режиме реального времени.
* Выявление аномалий и генерацию уведомлений.
* Удобный интерфейс для отображения данных и управления устройствами. Проект сочетает актуальные технологии (AWS, React, Node.js) и позволяет работать с симуляцией данных IoT, что делает его практичным для обучения и применения.

## **2. Область применения (Scope)**

Приложение может быть использовано в различных сферах:

* Умный дом (мониторинг температуры, освещения, влажности).
* Промышленность (контроль состояния оборудования).
* Здравоохранение (отслеживание показателей среды).

Основные функции:

* Сбор и хранение данных от IoT-устройств.
* Анализ данных и выявление аномалий.
* Генерация алертов и управление уведомлениями.
* Безопасный доступ для пользователей с ролями.

## **3. Требования к функционалу (Functional Requirements)**

### **3.1 Сбор данных:**

* UDP-имитатор отправляет данные устройств на ECS/Fargate.
* Данные передаются в AWS Data Stream (DS).

### **3.2 Обработка данных:**

* Lambda-функция анализирует данные, выявляет аномалии, создает алерты.
* Lambda-функция сохраняет данные в базу данных (MongoDB или DynamoDB).

### **3.3 Генерация уведомлений:**

* Lambda-функция отправляет уведомления через AWS SNS или push-систему.

### **3.4 Визуализация данных:**

* React-дашборд отображает:
  + Графики трендов данных.
  + Таблицы текущих показателей.
  + Список алертов и ошибок.

### **3.5 Управление пользователями:**

* Node.js-бэкенд обеспечивает:
  + Аутентификацию через JWT.
  + Ролевую авторизацию (администратор, оператор, пользователь).

## **4. Нефункциональные требования (Non-Functional Requirements)**

* **Безопасность:**
  + JWT хранится в HTTP Only Cookies для защиты данных.
  + Конфиденциальные ключи хранятся в AWS Secrets Manager.
* **Масштабируемость:**
  + Использование AWS ECS/Fargate для микросервисов.
  + Добавим параметры для настройки типов данных (температура, влажность, уровень освещения).
* **Производительность:**
  + Реакция интерфейса на запросы не дольше 3 секунд.
  + Сбор данных каждые 10 секунд.

## **5. Архитектура системы (System Architecture)**

### **Основные компоненты:**

1. Микросервисы:
   * AWS Lambda для обработки данных.
   * UDP-сервис на ECS/Fargate для симуляции данных.
2. Бэкенд:
   * Node.js (Express.js) для API и координации микросервисов.
3. Фронтенд:
   * React-приложение для отображения данных.
4. Сервисы AWS:
   * Data Streams, CloudWatch, SNS, Secrets Manager, S3.

## **6. Размышления по реализации (Implementation Things)**

* **Генерация данных:** Использование симулятора для имитации работы IoT-устройств (хотя было бы интересно связать с реальными устройствами). (UDP-имитатор будет отправлять данные о сенсорах в ECS/Fargate.)
* **Обработка аномалий:** Реализация алгоритмов анализа данных для выявления трендов (например, постепенного повышения температуры, постоянного ухудшения качества воздуха).
* **Визуализация:** Использование библиотек Chart.js и Redux для удобного интерфейса.
* **Деплой:** Использование CI/CD на GitHub Actions. Настройка автоматизации через AWS CloudFormation и SAM для быстрого развертывания.

## **7. Используемые технологии (Technologies)**

* **AWS:** Lambda, ECS/Fargate, CloudWatch, Data Streams, SNS, Secrets Manager.
* **Бэкенд:** Node.js, Express.js
* **Фронтенд:** React, Redux, Chart.js
* **База данных:** MongoDB (или DynamoDB).
* **Логирование:** Winston для интеграции с CloudWatch.

#### **Описание проекта**

**IoT Monitoring Platform** — это система для сбора, обработки и мониторинга данных от IoT-устройств (например, сенсоров температуры, влажности, освещенности и т.д.). Платформа предоставляет пользователям возможность отслеживать состояние устройств в реальном времени, просматривать исторические данные и получать уведомления о неисправностях или аномалиях в данных через удобный интерфейс.

Этот проект интересен, так как он сочетает в себе работу с облачными технологиями, микросервисами, безопасностью и визуализацией данных. Он также реалистичен, так как подобные системы востребованы в таких областях, как умный дом, промышленный мониторинг и здравоохранение.

#### **Основные функции**

1. **Сбор данных**: Микросервисы на AWS Lambda собирают данные от IoT-устройств и сохраняют их в базе данных.
2. **Обработка данных**: Микросервисы анализируют данные, выявляют аномалии (например, превышение пороговых значений) и генерируют алерты.
3. **Мониторинг**: React-фронтенд предоставляет дашборд с графиками, таблицами и списком алертов/ошибок.
4. **Управление пользователями**: Node.js-бэкенд управляет пользователями и их ролями, обеспечивает возможность настройки ролей.
5. **Безопасность**: Аутентификация через JWT, авторизация на основе ролей (администратор, оператор, пользователь).
6. **Логирование активности:** отслеживание активности пользователей (учет действий через логирование в базе данных).
7. **Логирование и обработка ошибок**: Интеграция с AWS CloudWatch для логирования и уведомлений, отображение ошибок на фронтенде.

#### **Соответствие требованиям**

##### **1. Архитектура (Architecture)**

* **Микросервисы**: Реализованы как AWS Lambda-функции на Java. Примеры микросервисов:
  + Сбор данных от IoT-устройств.
  + Обработка данных и выявление аномалий.
  + Генерация алертов.
* **Бэкенд**: Node.js-приложение с Express.js для предоставления API фронтенду и координации микросервисов.
* **Фронтенд**: React-приложение для визуализации данных и взаимодействия с пользователем.

##### **2. Обработка ошибок и логирование (Errors Handling and Logging)**

* **Логирование**: Приложение использует логи с уровнями (info, debug, warn, error, alert) через библиотеки Winston & Morgan, интегрированную с AWS CloudWatch.
* **Ошибки ядра приложения**: Исключения, связанные с неисправностью микросервисов, логируются с уровнем alert, настраиваются в CloudWatch как алерты с отправкой email/SMS техническому персоналу через AWS SNS.
* **Ошибки данных**: Исключения из-за несогласованности данных (например, некорректные показания сенсоров) логируются с уровнем error.
* **Дашборд ошибок**:
  + Реализован polling-подход:
    - Запланированная Lambda-функция периодически проверяет CloudWatch на наличие исключений и записывает их во внешнюю базу данных (например, MongoDB).
    - React-фронтенд через определенные интервалы опрашивает Node.js-бэкенд, который извлекает данные об ошибках из базы.

##### **3. Управление безопасностью (Security Management)**

* **Node.js**:
  + **Аутентификация**: Используется JWT для проверки подлинности пользователей.
  + **Авторизация**: Ролевая модель (например, администратор — полный доступ, оператор — мониторинг, пользователь — только свои устройства).
  + **Accounting**: Простое собственное решение для учета действий пользователей (например, логирование входов и операций в базу данных).
* **React**:
  + Страница входа отображается при старте сессии и при истечении JWT.
  + Доступ к страницам ограничен ролью пользователя (например, администратор видит дашборд ошибок, пользователь — только свои данные).
  + JWT хранится в **HTTP Only Cookies**, отправляемых бэкендом, для повышения безопасности (альтернатива — session storage).

##### **4. Конфигурация (Configuration)**

* **AWS**:
  + Переменные окружения для настройки Lambda-функций и других сервисов.
  + Чувствительные данные (например, ключи API, пароли) хранятся в **AWS Secrets Manager**.
* **React**:
  + Переменные окружения через .env-файлы для настройки API-URL и других параметров.
* **Node.js**:
  + Используется пакет config с файлами default.json и production.json для управления именами переменных окружения.
  + Значения переменных окружения задаются в зависимости от среды (dev/prod).

##### **5. Развертывание (Deployment)**

* **AWS Services**:
  + **CloudFormation**: Для описания инфраструктуры (Lambda, CloudWatch, S3 и т.д.).
  + **SAM**: Для упрощения развертывания серверных компонентов.
* **Back-Office**:
  + **Фронтенд**: Хостинг на S3.
  + **Бэкенд**: Развертывание на ECS/Fargate для гибкости и масштабируемости.

#### **Технологии**

* **AWS**: Lambda, CloudWatch, Secrets Manager, CloudFormation, SAM, S3, ECS/Fargate, SNS (для уведомлений).
* **Бэкенд**: Node.js, Express.js, MongoDB (или другая NoSQL-база для ошибок).
* **Фронтенд**: React, Redux (для управления состоянием), библиотеки для графиков (например, Chart.js).
* **Безопасность**: JWT, HTTP Only Cookies.
* **Логирование**: Winston, Morgan.

#### **Дополнительные детали**

* **Симуляция данных**: Для упрощения можно генерировать данные IoT-устройств вместо реального подключения (например, через AWS IoT Core).
* **Интерфейс**: Дашборд включает графики (тренды данных), таблицы (список устройств) и алерты (ошибки/аномалии).
* **Расширение**: Возможность добавить уведомления через push или интеграцию с умными устройствами.

#### **Почему этот проект?**

* **Интерес**: Работа с IoT актуальна и позволяет экспериментировать с реальными сценариями.
* **Реалистичность**: Подобные системы используются в промышленности и умных городах.
* **Соответствие требованиям**: Полностью охватывает архитектуру, безопасность, логирование и развертывание.

Структура и реализация проекта

## **1. Структура Проекта (Монорепозиторий)**

Предлагается структура монорепозитория для упрощения управления зависимостями и согласованности.

iot-monitoring-system/

├── backend/ # Node.js/Express Backend API

│ ├── src/

│ │ ├── controllers/ # Обработчики запросов (логика API)

│ │ ├── middleware/ # Промежуточное ПО (auth, logging, error handling)

│ │ ├── models/ # Модели данных (если используется ORM/ODM)

│ │ ├── routes/ # Определение маршрутов API

│ │ ├── services/ # Бизнес-логика (взаимодействие с БД, внешними сервисами)

│ │ ├── config/ # Управление конфигурацией (пакет 'config')

│ │ ├── utils/ # Вспомогательные функции

│ │ └── app.js # Инициализация Express приложения

│ ├── test/ # Модульные и интеграционные тесты

│ ├── Dockerfile # Docker-файл для сборки образа бэкенда

│ ├── package.json

│ └── ...

│

├── frontend/ # React Frontend

│ ├── public/

│ ├── src/

│ │ ├── components/ # Переиспользуемые UI-компоненты (графики, таблицы)

│ │ ├── pages/ # Основные страницы (Dashboard, Login, Devices, Errors)

│ │ ├── features/ # Redux slices / Логика фич (auth, data, devices, errors)

│ │ ├── services/ # Функции для вызова API бэкенда (axios)

│ │ ├── hooks/ # Пользовательские хуки React

│ │ ├── store/ # Настройка Redux store

│ │ ├── utils/ # Вспомогательные функции

│ │ ├── App.js # Основной компонент с роутингом

│ │ └── index.js # Точка входа

│ ├── .env.development # Переменные окружения для разработки

│ ├── .env.production # Переменные окружения для продакшена

│ ├── package.json

│ └── ...

│

├── lambdas/ # AWS Lambda функции (Java)

│ ├── data-processor/ # Лямбда для обработки данных из Kinesis

│ │ ├── src/main/java/com/example/iot/dataprocessor/

│ │ │ └── Handler.java # Основной обработчик лямбды

│ │ ├── pom.xml # Maven конфигурация (или build.gradle)

│ │ └── template.yaml # SAM шаблон для этой лямбды (опционально)

│ │

│ ├── alerter/ # Лямбда для отправки уведомлений

│ │ ├── src/main/java/com/example/iot/alerter/

│ │ │ └── Handler.java

│ │ ├── pom.xml

│ │ └── template.yaml

│ │

│ ├── error-poller/ # Лямбда для сбора ошибок из CloudWatch

│ │ ├── src/main/java/com/example/iot/errorpoller/

│ │ │ └── Handler.java

│ │ ├── pom.xml

│ │ └── template.yaml

│ │

│ └── common/ # (Опционально) Общий код для лямбд (модели, утилиты)

│ ├── src/main/java/com/example/iot/common/

│ ├── pom.xml

│

├── simulator/ # UDP Имитатор данных IoT

│ ├── src/ # Код симулятора (может быть Node.js, Python, Java)

│ ├── Dockerfile # Docker-файл для сборки образа симулятора

│ ├── package.json # (если Node.js)

│ └── ...

│

├── udp-listener/ # UDP Listener Service (на ECS/Fargate)

│ ├── src/ # Код сервиса (Node.js или Java)

│ ├── Dockerfile # Docker-файл для сборки образа

│ ├── package.json # (если Node.js)

│ └── ...

│

├── infra/ # Инфраструктура как код (IaC)

│ ├── main.yaml # Основной шаблон CloudFormation/SAM

│ ├── parameters/ # Файлы параметров для разных сред (dev, prod)

│ └── nested-stacks/ # (Опционально) Вложенные стеки для модульности

│

├── docs/ # Документация (API, архитектура)

│

├── scripts/ # Скрипты для сборки, деплоя, тестов

│ ├── deploy.sh

│ └── build.sh

│

├── .gitignore

└── README.md # Описание проекта, инструкции по запуску

## **2. Проработка Аспектов Реализации по Компонентам**

### **2.1 UDP Имитатор (simulator/)**

* **Технология:** Node.js (простота I/O)
* **Логика:**
  + Генерирует данные (JSON) для нескольких виртуальных устройств (Device ID).
  + Формат данных: { "deviceId": "UUID", "timestamp": ISO8601\_UTC, "type": "temperature|humidity|light", "value": number }.
  + Периодически (каждые 10 секунд, согласно NFR) отправляет UDP пакеты на адрес udp-listener сервиса.
  + Адрес и порт udp-listener берутся из переменных окружения.
* **Развертывание:** Docker-контейнер на AWS ECS/Fargate. Настраивается через сервис обнаружения ECS или получает адрес Load Balancer'а udp-listener.

### **2.2 UDP Listener Service (udp-listener/)**

* **Технология:** Node.js (используя dgram) или Java (используя DatagramSocket). Node.js предпочтительнее для легковесных сетевых сервисов.
* **Логика:**
  + Слушает указанный UDP порт.
  + При получении пакета:
    - Парсит JSON-данные.
    - Проводит базовую валидацию формата.
    - Использует AWS SDK для отправки записи (или батча записей) в AWS Kinesis Data Streams. Использует deviceId в качестве partitionKey для равномерного распределения.
  + Обрабатывает ошибки сети и отправки в Kinesis.
  + Логирует входящие данные и результаты отправки в CloudWatch (Winston).
* **Развертывание:** Docker-контейнер на AWS ECS/Fargate, вероятно, за Network Load Balancer (NLB) для UDP трафика.
* **IAM Роль:** Требует прав kinesis:PutRecord, kinesis:PutRecords, logs:CreateLogStream, logs:PutLogEvents.

### **2.3 AWS Kinesis Data Streams**

* **Настройка:** Создается через CloudFormation/SAM. Количество шардов зависит от ожидаемой нагрузки (начну с 1 для разработки). Настроить период хранения данных (Retention Period).

### **2.4 Data Processor Lambda (lambdas/data-processor/ - Java)**

* **Технология:** Java 21, Maven, AWS SDK for Java v2, Jackson (для JSON), SLF4j + Logback (для логирования).
* **Триггер:** AWS Kinesis Data Streams. Настроить размер батча и окно.
* **Логика:**
  + Получает батч записей из Kinesis.
  + Десериализует каждую запись из JSON в Java POJO (SensorData).
  + **Валидация:** Проверяет корректность типов, диапазонов значений.
  + **Анализ аномалий:**
    - Простые пороговые значения (конфигурируемые).
    - Анализ тренда (опционально): сравнение с предыдущими N значениями для того же deviceId (потребует чтения из БД или использования Kinesis Analytics). Начать с пороговых.
  + **Сохранение данных:**
    - Записывает обработанные данные (включая флаг аномалии, если есть) в базу данных.
    - **Выбор БД:**
      * **DynamoDB (Рекомендуется):** Отлично масштабируется, интегрируется с Lambda через IAM, модель оплаты по запросам. Подходит для больших объемов временных рядов. Ключ раздела (Partition Key): deviceId. Ключ сортировки (Sort Key): timestamp.
      * **MongoDB (Atlas/DocumentDB):** Более гибкие запросы, привычный интерфейс. Может потребовать управления подключением (connection pooling) в Lambda.
  + **Генерация алертов:** Если обнаружена аномалия, асинхронно вызывает alerter Lambda (LambdaClient.invoke с InvocationType.EVENT), передавая детали аномалии.
* **Логирование:** Подробное логирование в CloudWatch в JSON-формате с использованием Logback (например, через logstash-logback-encoder). Логировать начало/конец обработки батча, количество записей, найденные аномалии, ошибки.
* **Конфигурация:** Имена таблиц/коллекций БД, пороговые значения аномалий, ARN alerter Lambda - через переменные окружения, чувствительные данные (пароли БД, если не IAM auth) - через AWS Secrets Manager.
* **IAM Роль:** kinesis:GetRecords, kinesis:GetShardIterator, kinesis:DescribeStream, dynamodb:PutItem (или эквивалент для Mongo), lambda:InvokeFunction (для вызова alerter), secretsmanager:GetSecretValue, logs:CreateLogStream, logs:PutLogEvents.

### **2.5 Alerter Lambda (lambdas/alerter/ - Java)**

* **Технология:** Java, AWS SDK for Java v2 (SNS), Jackson, SLF4j + Logback.
* **Триггер:** Асинхронный вызов от data-processor.
* **Логика:**
  + Получает полезную нагрузку с деталями аномалии (deviceId, timestamp, type, value, anomaly description).
  + Формирует понятное сообщение для пользователя/администратора.
  + Публикует сообщение в настроенный AWS SNS Topic.
* **Логирование:** Логирует полученные данные и результат публикации в SNS.
* **Конфигурация:** ARN SNS-топика через переменные окружения.
* **IAM Роль:** sns:Publish, logs:CreateLogStream, logs:PutLogEvents.

### **2.6 Error Poller Lambda (lambdas/error-poller/ - Java)**

* **Технология:** Java, AWS SDK for Java v2 (CloudWatch Logs, DynamoDB/MongoDB Driver), Jackson, SLF4j + Logback.
* **Триггер:** Запланированное событие AWS EventBridge (e.g., rate(5 minutes)).
* **Логика:**
  + Использует CloudWatchLogsClient.filterLogEvents или startQuery/getQueryResults (CloudWatch Logs Insights) для поиска сообщений с уровнем ERROR в лог-группах других лямбд и бэкенд-сервиса за последний интервал времени.
  + Извлекает релевантную информацию (timestamp, log stream, message, stack trace).
  + Записывает уникальные ошибки (можно использовать хэш сообщения для дедупликации) в отдельную таблицу/коллекцию errors в DynamoDB/MongoDB. Добавляет поля status ('new'), timestampFirstSeen, timestampLastSeen, count.
* **Логирование:** Старт/конец выполнения, количество найденных/записанных ошибок.
* **Конфигурация:** Имена лог-групп для мониторинга, имя таблицы/коллекции ошибок, параметры подключения к БД - через переменные окружения/Secrets Manager.
* **IAM Роль:** logs:FilterLogEvents, logs:StartQuery, logs:GetQueryResults, dynamodb:PutItem, dynamodb:UpdateItem (или эквивалент для Mongo), secretsmanager:GetSecretValue, logs:CreateLogStream, logs:PutLogEvents.

### **2.7 Backend API (backend/ - Node.js/Express.js)**

* **Технология:** Node.js (LTS), Express.js, jsonwebtoken, bcryptjs, cookie-parser, cors, helmet, winston, config, AWS SDK for JavaScript v3 (для Secrets Manager, DynamoDB), mongodb (если используется MongoDB).
* **Маршруты (routes/):**
  + auth: /register, /login, /logout, /me (получение текущего пользователя).
  + users: CRUD для пользователей (доступ для admin).
  + devices: CRUD для устройств (доступ для admin, возможно operator). Связь устройств с пользователями (если требуется).
  + data: /latest?deviceId=..., /historical?deviceId=...&from=...&to=... (чтение из БД данных сенсоров).
  + alerts: /alerts?deviceId=...&status=... (чтение алертов, вероятно, сохраненных вместе с данными или в отдельной коллекции).
  + errors: /errors?status=..., /errors/:id/acknowledge (чтение и обновление статуса ошибок из БД errors).
* **Middleware (middleware/):**
  + authenticate: Проверяет JWT из HttpOnly cookie. Добавляет req.user.
  + authorize(role): Проверяет req.user.role против требуемой роли.
  + requestLogger: Логирование запросов с помощью Winston.
  + errorHandler: Централизованная обработка ошибок.
* **Аутентификация/Авторизация:**
  + Регистрация: Хэширование пароля (bcryptjs?).
  + Логин: Сравнение хэша пароля, генерация JWT, установка JWT в HttpOnly, Secure, SameSite=Strict cookie.
  + Роли: admin, operator, user. Хранятся в записи пользователя в БД.
* **Учет действий (Accounting):** В middleware или контроллерах логировать важные действия (логин, изменение данных) с указанием пользователя в CloudWatch Logs или отдельную коллекцию БД (activity\_logs).
* **База данных:** Настроить подключение к DynamoDB/MongoDB. Использовать AWS SDK или ODM (Mongoose для MongoDB).
* **Конфигурация (config/):** Использовать пакет config. Определить default.json, production.json, development.json. Читать имена переменных окружения из конфига. Значения (JWT Secret, DB connection string/credentials) получать из переменных окружения, которые в свою очередь в ECS Task Definition берутся из AWS Secrets Manager.
* **Логирование:** Winston, настроенный на вывод в консоль (для ECS -> CloudWatch) в JSON-формате. Уровни info, warn, error.
* **Развертывание:** Docker-контейнер на AWS ECS/Fargate за Application Load Balancer (ALB).
* **IAM Роль:** secretsmanager:GetSecretValue, dynamodb:\* / права для MongoDB, logs:CreateLogStream, logs:PutLogEvents.

### **2.8 Frontend (frontend/ - React)**

* **Технология:** React, Redux Toolkit, React Router, Axios, Chart.js (react-chartjs-2), date-fns или dayjs.
* **Управление состоянием (Redux Toolkit):**
  + authSlice: Статус аутентификации, информация о пользователе, роль.
  + deviceSlice: Список устройств.
  + dataSlice: Данные для графиков и таблиц, статус загрузки.
  + alertSlice: Список алертов.
  + errorSlice: Список ошибок с дашборда ошибок.
* **Маршрутизация (React Router):**
  + Публичный маршрут: /login.
  + Приватные маршруты: /dashboard, /devices, /users (admin), /errors (admin/operator). Использовать обертку для проверки аутентификации и роли пользователя из authSlice.
* **Взаимодействие с API:** Использовать axios. Создать инстанс axios с baseURL и withCredentials: true для автоматической отправки cookies. Обрабатывать ответы и ошибки, обновлять Redux store.
* **Визуализация:**
  + Chart.js: Для отображения графиков трендов (Line chart).
  + Таблицы: Для текущих показателей, списка устройств, алертов, ошибок.
* **Обновление данных:**
  + Для "реального времени": Периодический опрос API (polling) с помощью setInterval / setTimeout в useEffect. Интервал настроить (e.g., 5-15 секунд).
  + Для исторических данных: Запрос при выборе диапазона дат.
* **Безопасность:** Не хранить JWT в localStorage или sessionStorage. Бэкенд должен устанавливать HttpOnly cookie. React просто делает запросы. Обрабатывать редирект на /login при 401/403 ответах API (кроме самой страницы логина).
* **Конфигурация:** URL бэкенд-API через переменные окружения (REACT\_APP\_API\_URL в .env файлах).
* **Развертывание:** Сборка статических файлов (npm run build). Загрузка в S3 bucket. Настройка CloudFront для раздачи, HTTPS и кеширования.

## **3. База Данных**

* **Данные Устройств (Высокая Нагрузка на Запись):** **DynamoDB** - основной кандидат.
  + Таблица DeviceData: deviceId (Partition Key, String), timestamp (Sort Key, Number - Unix epoch или ISO8601 String), value (Number), type (String), isAnomaly (Boolean).
  + Таблица Devices: deviceId (Partition Key, String), name (String), location (String), ownerUserId (String, опционально), thresholds (Map, опционально).
* **Пользователи, Роли, Конфигурация Устройств:** **DynamoDB** или **MongoDB (Atlas/DocumentDB)**.
  + Таблица/Коллекция Users: userId (PK), username, passwordHash, role (admin, operator, user).
* **Ошибки Приложения:** **MongoDB (Atlas/DocumentDB)** или **DynamoDB**.
  + Таблица/Коллекция ApplicationErrors: errorId (PK), timestampFirstSeen, timestampLastSeen, source (e.g., lambda name, backend), message, stackTrace, status ('new', 'acknowledged'), count. MongoDB может быть удобнее для поиска по тексту ошибок.
* **Логи Активности:** **CloudWatch Logs** или отдельная таблица/коллекция (если нужен сложный поиск).

## **4. Безопасность**

* **Аутентификация:** JWT, передаваемый через HttpOnly, Secure, SameSite=Strict cookies.
* **Авторизация:** Ролевая модель, проверяемая на бэкенде (middleware).
* **Секреты:** ВСЕ секреты (JWT secret key, пароли БД, ключи API) хранятся в **AWS Secrets Manager**. Доступ к ним осуществляется через IAM роли, назначенные Lambda-функциям и ECS задачам.
* **Сетевая Безопасность:** Настройка Security Groups для ограничения трафика между компонентами (ECS, Lambda, DB). Использовать VPC Endpoints для доступа к сервисам AWS (Kinesis, DynamoDB, Secrets Manager) без выхода в публичный интернет.
* **Защита API:** Helmet для установки заголовков безопасности (?), CORS для контроля доступа с фронтенда (?), возможное добавление rate limiting.
* **Защита Фронтенда:** HTTPS через CloudFront.

## **5. Логирование и Мониторинг**

* **Логирование:**
  + Стандарт: JSON формат для всех логов.
  + Инструменты: Winston, Morgan (Node.js), SLF4j + Logback с JSON энкодером (Java).
  + Централизация: Все логи отправляются в **AWS CloudWatch Logs**.
  + Уровни: info, debug, warn, error, alert.
* **Мониторинг:**
  + **CloudWatch Metrics:** Стандартные метрики Lambda (invocations, duration, errors), Kinesis (PutRecord latency, GetRecords iterator age), ECS (CPU/Memory), ALB (request count, latency, errors), DynamoDB (Consumed RCU/WCU, throttles).
  + **CloudWatch Alarms:** Настроить алармы на основе метрик (e.g., Lambda errors > 0, Kinesis IteratorAge > 50% retention period, высокий % ошибок ALB, троттлинг DynamoDB).
  + **CloudWatch Dashboards:** Создать дашборды для визуализации ключевых метрик и состояния системы.
* **Уведомления об Ошибках Ядра:** CloudWatch Alarms -> SNS Topic -> Email/SMS для технического персонала; логирование с уровнем alert.
* **Дашборд Ошибок Приложения:** Реализован через error-poller Lambda -> DB -> Backend API -> React Frontend.

## **6. Конфигурация**

* **AWS Lambda/ECS:** Переменные окружения, значения которых (особенно секреты) подтягиваются из AWS Secrets Manager или Parameter Store.
* **Node.js Backend:** Пакет config, переменные окружения, интеграция с AWS Secrets Manager через AWS SDK.
* **React Frontend:** .env файлы (REACT\_APP\_...).

## **7. Развертывание (CI/CD)**

* **Инфраструктура:** AWS CloudFormation или SAM, управляемые через Git.
* **Backend/Simulator/Listener:**
  1. Сборка Docker-образа.
  2. Публикация образа в AWS ECR.
  3. Обновление ECS Task Definition и Service (через CloudFormation/SAM или AWS CLI/SDK).
* **Lambdas (Java):**
  1. Сборка JAR (mvn package).
  2. Упаковка с помощью aws cloudformation package / sam package.
  3. Развертывание с помощью aws cloudformation deploy / sam deploy.
* **Frontend:**
  1. Сборка статики (npm run build).
  2. Синхронизация с S3 bucket (aws s3 sync).
  3. (Опционально) Инвалидация кеша CloudFront (aws cloudfront create-invalidation).
* **Автоматизация:** Использовать AWS CodePipeline, GitHub Actions или Jenkins для автоматизации этих шагов при коммитах в Git (GitHub Actions workflow основной кандидат).

# Экономическая модель проекта.

### Выбор NoSQL DB Engine (MongoDB vs DynamoDB)

Аргументация в пользу DynamoDB как основного кандидата для хранения данных устройств в этом сценарии строится на нескольких ключевых факторах, особенно при рассмотрении **высокой нагрузки на запись** и **общей архитектуры**, а не только на абсолютной стоимости хранения гигабайта данных.

Вот развернутые аргументы:

1. **Модель Ценообразования и Масштабируемость Записи:**

* **DynamoDB (On-Demand):** Вы платите за фактические операции чтения и записи (Read/Write Capacity Units - RCU/WCU). В режиме "On-Demand" DynamoDB *автоматически и мгновенно* масштабируется для обработки практически любой нагрузки на запись без необходимости ручного вмешательства или предварительного прогрева. Это идеально для IoT, где нагрузка может быть непредсказуемой или пиковой. Вы платите только за то, что используете. Нет платы за простаивающие ресурсы.
* **MongoDB Atlas:** Ценообразование в основном зависит от размера кластера (vCPU, RAM, Storage), который вы *провизионируете*. Чтобы справиться с высокой нагрузкой на запись, вам нужно заранее выбрать достаточно мощный кластер (или настроить автоскейлинг, который реагирует медленнее и сложнее, чем у DynamoDB On-Demand). Если ваша пиковая нагрузка на запись высока, но непостоянна, вы будете платить за большой кластер Atlas, даже когда нагрузка низкая. Да, есть Serverless-инстансы Atlas, но они новее и их модель ценообразования/производительности может отличаться, и они все еще могут быть не так оптимизированы для мгновенных пиков записи, как DynamoDB.
* **Вывод по цене/масштабированию:** Для *переменной* или *пиковой* нагрузки на запись, характерной для многих IoT-сценариев, модель DynamoDB On-Demand часто оказывается *экономически эффективнее в целом*, так как вы не переплачиваете за неиспользуемые ресурсы в периоды низкой активности. Atlas может быть дешевле при *очень высокой и стабильной* нагрузке, где вы можете точно подобрать и постоянно утилизировать ресурсы кластера.

1. **Серверная (Serverless) Природа и Интеграция с Lambda:**

* **DynamoDB:** Это полностью управляемый, серверный сервис. Он идеально интегрируется с AWS Lambda. Не нужно управлять пулами соединений из Lambda (что является частой проблемой при работе с традиционными БД из лямбд), используется нативная IAM-аутентификация. Масштабирование DynamoDB не влияет на производительность или конфигурацию Lambda.
* **MongoDB Atlas:** Хотя Atlas — это управляемый сервис, он все же представляет собой кластер серверов. При работе с Lambda вам нужно будет управлять подключениями (использовать библиотеки для connection pooling или официальный Data API, который может вносить свою задержку/стоимость). Интеграция через IAM менее нативна. Масштабирование кластера Atlas не связано напрямую с масштабированием Lambda.
* **Вывод по интеграции:** Бесшовная, серверная интеграция DynamoDB с Lambda упрощает разработку, повышает надежность и снижает операционные издержки, что косвенно влияет на общую стоимость владения.

1. **Операционные Накладные Расходы:**

* **DynamoDB:** Почти нулевые операционные расходы. Нет серверов для патчинга, нет ОС для обновления, нет необходимости управлять репликацией или шардированием вручную (оно происходит под капотом).
* **MongoDB Atlas:** Управляемый сервис снимает *большую часть* операционных забот, но вам все еще может потребоваться принимать решения о версиях MongoDB, стратегиях шардирования (если требуется), настройках резервного копирования, выборе и изменении размеров инстансов.
* **Вывод по операциям:** Меньшие операционные расходы DynamoDB экономят время команды, что транслируется в снижение общей стоимости.

1. **Оптимизация для Шаблона Доступа IoT:**

* Типичный шаблон для данных IoT — это запись большого количества небольших сообщений, индексированных по deviceId и timestamp. DynamoDB отлично подходит для этого шаблона "ключ-значение" с возможностью запросов по диапазону (по временной метке с использованием Sort Key).
* MongoDB более гибок в запросах, что полезно для аналитики, но эта гибкость может быть избыточной (и потенциально более дорогой) для основной задачи — быстрой записи и простого извлечения временных рядов.

Учитывая требования SRS (сбор данных в реальном времени, Lambda для обработки, масштабируемость), **DynamoDB выглядит более естественным и потенциально более экономичным выбором именно для задачи хранения потока данных от устройств** из-за его серверной природы, модели оплаты по запросу (On-Demand) и бесшовной интеграции с экосистемой AWS, особенно с Lambda. Для других данных (пользователи, ошибки) MongoDB может быть вполне разумным выбором, если его гибкость там нужнее.

### Использование реляционной БД

рассмотрим использование реляционной SQL-базы данных, в частности PostgreSQL (например, через AWS RDS for PostgreSQL), для тех частей проекта, где ранее рекомендовался MongoDB (данные пользователей, конфигурация устройств, ошибки приложения, логи активности).

**Аргументы в пользу PostgreSQL для этих данных:**

1. **Реляционная Целостность (Users, Devices, Activity Logs):**

* **Связи:** Данные пользователей, устройств и логов активности بطبيعتها (by nature) реляционны. Пользователь владеет устройствами, логи активности ссылаются на пользователей и, возможно, на устройства.
* **PostgreSQL:** Позволяет определить внешние ключи (Foreign Keys) для строгого обеспечения этих связей на уровне базы данных. Это предотвращает появление "осиротевших" записей (например, лог активности без пользователя или устройство без владельца) и гарантирует целостность данных.
* **MongoDB:** Требует поддержания этих связей на уровне приложения, что увеличивает сложность кода и риск возникновения несогласованных данных.

1. **Транзакционная Надежность (ACID):**

* **PostgreSQL:** Предоставляет надежные ACID-транзакции "из коробки". Это важно для операций, требующих атомарности, например, регистрация пользователя (создание записи пользователя + назначение роли) или обновление статуса ошибки.
* **MongoDB:** Поддерживает транзакции, но они исторически были более сложными в реализации или имели другие характеристики производительности по сравнению с традиционными SQL СУБД. Для простых операций это может быть избыточно, но для сложных – важно.

1. **Мощность и Стандартность SQL:**

* **PostgreSQL:** Использует стандартный язык SQL, который чрезвычайно мощен для выполнения сложных запросов, агрегаций, соединений (JOINs). Получить список пользователей с их устройствами или отчет по ошибкам за период с группировкой по типу – это стандартные SQL-запросы.
* **MongoDB:** MQL (MongoDB Query Language) гибок, но не стандартизирован. Операции, аналогичные JOIN ($lookup), могут быть менее производительными или интуитивными для сложных связей. SQL часто более знаком широкому кругу разработчиков.

1. **Работа с Полуструктурированными Данными (JSONB):**

* **PostgreSQL:** Обладает превосходной поддержкой типов данных JSON и JSONB. Тип JSONB (бинарный JSON) особенно эффективен, так как позволяет не только хранить JSON-документы внутри реляционной таблицы, но и **эффективно индексировать и запрашивать данные *внутри* этих JSON-документов**.
* **Применение:** Это идеально подходит для:
* **Конфигурации устройств:** Хранение специфичных порогов или настроек в поле config JSONB.
* **Деталей ошибок:** Хранение полного стектрейса или переменных контекста в поле details JSONB.
* **Логи активности:** Хранение дополнительных параметров действия в поле payload JSONB.
* Это дает гибкость, сравнимую с MongoDB, но в рамках строго типизированной реляционной модели с возможностями SQL.

1. **Зрелость и Экосистема:**

* PostgreSQL — очень зрелая, стабильная и расширяемая СУБД с огромной экосистемой инструментов, библиотек, ORM (Sequelize, TypeORM для Node.js; Hibernate, JOOQ для Java) и обширным сообществом.

**Потенциальные контраргументы (и их оспаривание):**

* **Жесткость схемы?** Традиционно SQL требует строгой схемы. Однако:
* Для пользователей, ролей, основных атрибутов устройств структура довольно стабильна.
* Гибкость для изменяющихся частей достигается через JSONB.
* Миграции схемы – это управляемый процесс, обеспечивающий контролируемое развитие структуры данных.
* **Масштабируемость?** Горизонтальное масштабирование PostgreSQL традиционно сложнее, чем у некоторых NoSQL баз. Однако:
* Объем данных пользователей, ошибок или логов активности редко достигает таких масштабов, как сырые данные с IoT устройств.
* Вертикального масштабирования (увеличение мощности инстанса RDS) и использования реплик чтения часто достаточно надолго.
* AWS RDS упрощает управление репликами и высокой доступностью.
* Для экстремальных случаев существуют решения вроде Citus Data (расширение для PostgreSQL).
* **Стоимость?** Инстанс RDS может иметь более высокую минимальную стоимость, чем небольшая таблица DynamoDB или маленький кластер Atlas. Однако нужно сравнивать общую стоимость владения (TCO), включая затраты на разработку (простота SQL, ORM) и поддержку целостности данных (встроенные механизмы PostgreSQL).

**Сравнение с MongoDB для этих задач:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Аспект** | **PostgreSQL (с JSONB)** | **MongoDB** | **Вывод для Users/Errors/Logs** |
| **Реляционная Целостность** | Встроенная (Foreign Keys) | На уровне приложения | **PostgreSQL +** |
| **Транзакции (ACID)** | Надежные, стандартные | Доступны, могут быть сложнее/менее привычны | **PostgreSQL +** |
| **Язык запросов** | SQL (мощный, стандартный, JOINs) | MQL (гибкий, NoSQL-специфичный, $lookup) | **PostgreSQL +** (для реляц. данных) |
| **Гибкость схемы** | Умеренная + высокая для JSONB полей | Высокая (для всего документа) | PostgreSQL достаточен с JSONB |
| **Работа с JSON** | Отличная (JSONB, индексация, запросы внутри JSON) | Нативная (основной формат) | Оба сильны, PostgreSQL удивляет |
| **Масштабируемость** | Вертикальная + реплики (легко), Шардинг (сложнее) | Горизонтальная (шардинг) проще концептуально | Для этого объема данных - паритет |
| **Операционные расходы (RDS)** | Низкие (управляемый сервис) | Низкие (Atlas - управляемый сервис) | Паритет |

**Заключение:**

Использование **PostgreSQL (например, через AWS RDS)** является **очень сильным и, возможно, даже предпочтительным** вариантом для хранения данных пользователей, конфигураций устройств, ошибок приложения и логов активности в данном проекте.

* Он обеспечивает **лучшую гарантию целостности данных** благодаря реляционной модели и ACID-транзакциям.
* **SQL** предоставляет мощные и стандартные средства для запросов и анализа этих данных.
* Поддержка **JSONB** эффективно решает проблему хранения и запроса полуструктурированных данных (детали ошибок, кастомные настройки), предлагая гибкость, сравнимую с MongoDB, без потери реляционных преимуществ.
* Вопросы масштабируемости для *этого типа данных* обычно легко решаются стандартными средствами PostgreSQL и управляемых сервисов вроде RDS.

Таким образом, архитектура может выглядеть так:

* **DynamoDB:** Для высоконагруженного потока сырых данных с устройств (временные ряды).
* **PostgreSQL (RDS):** Для метаданных устройств, пользователей, ролей, ошибок приложения, логов активности.

Это создает сбалансированную систему, использующую сильные стороны каждой технологии там, где они наиболее применимы.

1. **Модели Ценообразования:**

* **MongoDB Atlas:**
* **Shared Clusters (M0 Free, M2, M5):** Самые дешевые варианты для старта, разработки или приложений с очень низкой нагрузкой. Ограничены по ресурсам (CPU, RAM, IOPS, соединения), производительность может быть нестабильной ("шумные соседи"). *Часто самый дешевый способ начать.*
* **Dedicated Clusters (M10+):** Вы платите за выделенные ресурсы (vCPU, RAM, Storage). Цена растет с размером кластера. Доступно авто-масштабирование, но оно обычно масштабирует весь уровень кластера, что может быть дорогим шагом. Минимум 3 узла для репликации (HA). *Начальная цена M10 выше, чем у малых RDS инстансов.*
* **Serverless Instances:** Модель оплаты, похожая на DynamoDB On-Demand – плата за операции чтения/записи (Read/Write Processing Units) и хранение. Предназначена для непредсказуемых или нечастых нагрузок. *Может быть экономичной при очень переменной нагрузке, но нужно тщательно считать юниты и сравнивать с провижинированными вариантами.*
* **AWS RDS for PostgreSQL:**
* **On-Demand Instances:** Вы платите почасовую ставку за выбранный тип инстанса (e.g., db.t3.micro, db.m5.large, db.r6g.xlarge) + за хранилище + за IOPS (если применимо). Очень гранулярный выбор инстансов.
* **Reserved Instances:** Скидка за долгосрочные обязательства (1 или 3 года).
* **Storage:** Разные типы (gp2, gp3, io1, io2). gp3 особенно интересен, так как позволяет *независимо* настраивать объем хранилища, базовую производительность (IOPS) и пропускную способность, что дает большой потенциал для оптимизации затрат. Вы платите за объем (GB) и за provisioned IOPS/throughput сверх базовых.
* **High Availability (Multi-AZ):** Удваивает стоимость вычислительных ресурсов и хранилища, так как создается синхронный резервный инстанс в другой зоне доступности.

1. **Сравнение по Компонентам Стоимости:**

* **Compute (CPU/RAM):**
* На *низком* уровне нагрузки Atlas Shared (M2/M5) может быть дешевле, чем самый маленький RDS инстанс. Но RDS предлагает более предсказуемую производительность.
* На *среднем и высоком* уровне: RDS дает больше гибкости в выборе инстанса, позволяя точнее подобрать ресурсы под нагрузку. Часто можно найти RDS инстанс, который будет дешевле, чем эквивалентный по ресурсам *выделенный* кластер Atlas (M10+), у которого выше минимальный порог входа. Serverless Atlas может конкурировать, если нагрузка пиковая.
* **Storage:**
* Стоимость за GB обычно сопоставима между Atlas и RDS (особенно с gp3).
* Ключевое преимущество RDS gp3 – возможность платить только за необходимые IOPS, не переплачивая за лишний объем диска (в отличие от gp2 или некоторых моделей Atlas, где IOPS привязаны к размеру). Для баз данных с умеренным объемом, но требующих хорошего отклика (как база пользователей/ошибок), это может быть выгодно.
* **High Availability (HA):**
* Atlas (Dedicated): HA встроено через 3+ узла репликации. Стоимость включена в цену кластера.
* RDS: Multi-AZ удваивает стоимость инстанса и хранилища.
* Сравнение: Нужно считать напрямую. 3 узла Atlas M10 могут стоить сравнимо или дороже, чем 2 инстанса RDS (1 основной + 1 standby) эквивалентного размера.
* **Backups:** Стоимость хранения бэкапов обычно сопоставима и зависит от объема и срока хранения.
* **Data Transfer:** Внутри AWS затраты обычно минимальны или отсутствуют для трафика в пределах одного региона/VPC. Значимых различий между Atlas (на AWS) и RDS обычно нет.

1. **Косвенные Экономические Факторы:**

* **Операционные Расходы:** Оба сервиса управляемые, что сильно снижает операционные затраты по сравнению с self-hosted. RDS, как нативный сервис AWS, может предложить чуть более гладкую интеграцию с IAM, биллингом, CloudWatch в рамках единой экосистемы.
* **Стоимость Разработки/Поддержки:** Если ваша команда хорошо знает SQL и экосистему PostgreSQL (ORM, инструменты), разработка может идти быстрее и проще, что экономит деньги. Использование JSONB в Postgres позволяет достичь гибкости MongoDB там, где это нужно, не вводя вторую технологию БД для этих данных. Если же команда – эксперты MongoDB, выбор Atlas может быть экономически оправдан с точки зрения скорости разработки.

**Выводы по Экономике (MongoDB Atlas vs RDS PostgreSQL для Users, Configs, Errors, Logs):**

1. **Для Старта/Низкой Нагрузки:** Atlas Shared Tiers (M0/M2/M5) *могут быть* абсолютно дешевле самых маленьких RDS инстансов. RDS Free Tier может покрыть первый год.
2. **Для Стабильной Средней Нагрузки:** RDS часто выигрывает за счет более гранулярного выбора инстансов и оптимизируемого хранилища (gp3), позволяя точнее подобрать ресурсы и не переплачивать.
3. **Для Непредсказуемой/Пиковой Нагрузки:** Serverless Atlas является интересным вариантом и *может* быть дешевле, чем постоянно работающий (и часто недогруженный) RDS инстанс. Требует тщательного расчета стоимости юнитов.
4. **Простота и Гибкость RDS:** Возможность использовать одну технологию (PostgreSQL с JSONB) для разных типов структурированных и полуструктурированных данных (кроме основной time-series) может снизить сложность и, следовательно, косвенные затраты на разработку и поддержку по сравнению с использованием и PostgreSQL/SQL и MongoDB.
5. **Рекомендация:**

* Получите *оценку* ожидаемой нагрузки и объема данных для пользователей, ошибок и т.д.
* Используйте официальные калькуляторы AWS Pricing Calculator (для RDS) и MongoDB Atlas Pricing Calculator.
* Сравните стоимость:
* Небольшого RDS инстанса (e.g., db.t3.small или medium) с gp3 хранилищем (+ Multi-AZ если нужна HA).
* Atlas Dedicated кластера (e.g., M10 или M20) с нужным хранилищем.
* Atlas Serverless с вашей оценкой RPU/WPU.
* Учтите косвенные факторы (знакомство команды, сложность стека).

Часто для таких задач, как управление пользователями, конфигурациями и ошибками в типичном веб-приложении (даже бэкенде для IoT-дашборда), **RDS PostgreSQL оказывается достаточно экономичным и предсказуемым вариантом**, особенно с учетом преимуществ реляционной модели и мощи JSONB.