Technical Specification: Telegram Store MVP

Document Version: 1.0
Publication Date: 2025-11-30

1. Executive Summary

This document provides a comprehensive technical specification for the Minimum Viable Product (MVP) of a Telegram-based e-commerce store. The primary objective of this project is to establish a fully functional retail channel within the Telegram ecosystem, enabling users to browse products, manage a shopping cart, and complete orders directly through a bot interface. Concurrently, the system will provide administrators with a web-based panel for managing the product catalog, overseeing orders, and handling basic operational tasks.

The proposed solution is architected around a modern, scalable, and decoupled technology stack. The core of the system is a backend API developed using the **FastAPI** framework, chosen for its high performance, asynchronous capabilities, and robust data validation powered by Pydantic. This API will serve as the central nervous system, orchestrating all business logic and communication between the various components.

The user-facing component will be a **Telegram Bot** built with the **aiogram 3** library, leveraging its advanced features for state management (FSM), interactive inline keyboards, and efficient update handling via webhooks. This ensures a responsive and intuitive user experience for customers navigating the store.

For data persistence, a **PostgreSQL** database will be employed. The database schema is designed to be highly flexible, supporting complex product structures with multiple variants (e.g., size, color) and their corresponding attributes, prices, and stock levels. This relational model will also manage user data, shopping carts, and a complete order history.

Administrative functions will be facilitated through a **React** single-page application (SPA). This admin panel will communicate with the backend API via a secure, token-based authentication system using JSON Web Tokens (JWT) with a refresh token mechanism to ensure both security and a seamless user session. The panel will empower administrators to perform critical tasks, including product and order management, as well as data import and export using CSV/XLSX formats.

Key integrations are planned to streamline operations and provide valuable business insights. The system will integrate with the **Google Sheets API** for bulk product catalog management and order data exportation. For analytics, events such as product views, cart additions, and purchases will be tracked and sent to both **Google Analytics (GA4)** and **Yandex.Metrika**, providing a dual-source, comprehensive view of user behavior and e-commerce performance.

Security is a cornerstone of the architecture. Best practices will be implemented across all layers, including secure storage of credentials using environment variables and secrets management tools, rigorous input validation, and adherence to personal data protection regulations such as the Russian Federal Law 152-ФЗ. The architecture is designed for initial performance and future scalability, with clear pathways for optimization and expansion outlined in the post-MVP roadmap. This specification details

the functional and non-functional requirements, system architecture, data models, API contracts, and implementation guidelines necessary to successfully deliver the Telegram store MVP.

2. Requirements

This section outlines the functional and non-functional requirements for the MVP, along with user stories that illustrate the intended interactions from the perspective of both customers and administrators.

2.1. Functional Requirements

Customer-Facing (Telegram Bot):
* **Product Catalog:** Users must be able to view a list of product categories and browse products within those categories.
* **Product Details:** Users must be able to view detailed information for a single product, including its description, price, and available images or media.
* **Product Variants:** For products with variations (e.g., size, color), users must be able to view and select from the available options. The price and availability may change based on the selected variant.
* **Pagination:** Product and category lists that exceed a certain length must be paginated to ensure usability within the Telegram interface.
* **Shopping Cart:** Users must be able to add specific products or product variants to a shopping cart, view the contents of their cart, adjust quantities, and remove items.
* **Checkout Process:** Users must be able to initiate a checkout process from their cart, providing necessary information for order fulfillment (e.g., name, contact phone number, delivery address).
* **Order Placement:** Users must be able to confirm and place an order. Upon successful placement, they should receive a confirmation message with an order summary.
* **Order Notifications:** Users should receive notifications regarding significant changes in their order status (e.g., “Shipped,” “Delivered”).

Administrator-Facing (Web Panel):
* **Secure Authentication:** Administrators must log in to the admin panel through a secure authentication system.
* **Dashboard:** Upon login, administrators should see a dashboard with key metrics, such as the number of new orders and total sales volume for a given period.
* **Product Management:** Administrators must have full CRUD (Create, Read, Update, Delete) capabilities for products and their variants, including managing names, descriptions, prices, stock levels, and attributes.
* **Order Management:** Administrators must be able to view a list of all orders, filter them by status or date, and view the detailed information for each order. They must be able to update the status of an order (e.g., from “Pending” to “Processing” to “Shipped”).
* **Data Import/Export:** Administrators must be able to import a product catalog from a CSV or XLSX file. They must also be able to export order data to a CSV, XLSX, or Google Sheets document.
* **User Management:** Administrators must be able to view a list of customers who have placed orders.

2.2. Non-Functional Requirements

- **Security:** The system must protect sensitive user and order data. All communication between the admin panel and the backend must be encrypted (HTTPS). API keys, database credentials, and other secrets must be stored securely and not hardcoded in the application source. The system must comply with personal data protection laws, specifically 152-ФЗ, requiring the storage and processing of Russian citizens’ personal data on servers located in Russia.

- **Performance:** The Telegram bot must respond to user commands within 1-2 seconds for typical operations. The admin panel pages should load within 3 seconds. The backend API must handle concurrent requests efficiently.
- **Scalability:** The architecture should be designed to handle a growing number of users, products, and orders. The backend should be stateless to allow for horizontal scaling. The database schema should be optimized for query performance.
- **Reliability:** The system should be highly available, with minimal downtime. The webhook endpoint for the Telegram bot must be robust and handle potential connection issues or retries from Telegram's servers.
- **Usability:** The Telegram bot interface must be intuitive and easy to navigate for non-technical users. The admin panel should have a clean, organized layout that makes management tasks straightforward.
- **Maintainability:** The codebase must be well-structured, modular, and documented to facilitate future development and maintenance. A clear separation of concerns between the bot, backend API, and admin panel is required.

2.3. User Stories

Customer Stories:
* As a new user, I want to start the bot and see a welcome message with main menu options, so that I can understand how to begin shopping.
* As a customer, I want to browse products by category, so that I can easily find items I am interested in.
* As a customer, I want to view detailed information and images for a product, so that I can make an informed purchase decision.
* As a customer, I want to select a specific size and color for a T-shirt, so that I can order the exact variant I need.
* As a customer, I want to add multiple items to my shopping cart, so that I can purchase them all in a single transaction.
* As a customer, I want to view my cart and change the quantity of an item, so that I can correct my order before checkout.
* As a customer, I want to provide my name, phone number, and address during checkout, so that the store can deliver my order correctly.
* As a customer, I want to receive a confirmation message with my order number after placing an order, so that I have a record of my purchase.

Administrator Stories:
* As an administrator, I want to log in to a secure admin panel, so that I can manage the store's operations.
* As an administrator, I want to see a list of new orders on my dashboard, so that I can quickly begin processing them.
* As an administrator, I want to add a new product with multiple variants, defining a unique price and SKU for each, so that I can expand the store's catalog.
* As an administrator, I want to update the stock level of a product variant, so that the bot shows accurate availability to customers.
* As an administrator, I want to view all details of a specific order, including customer information and items purchased, so that I can fulfill it accurately.
* As an administrator, I want to change an order's status from "Processing" to "Shipped," so that the system and potentially the customer are updated on the progress.
* As an administrator, I want to upload a single XLSX file to add or update hundreds of products at once, so that I can manage the catalog efficiently.

* As an administrator, I want to export the last month’s orders to a Google Sheet, so that I can perform external analysis and reporting.

3. System Architecture

The system is designed with a modern, decoupled architecture to ensure scalability, maintainability, and a clear separation of concerns. It consists of several key components that interact through well-defined interfaces.

3.1. Components

The architecture is composed of the following primary components:

- **Telegram Bot (Client Interface):** This is the primary user-facing component, implemented as a Telegram bot using the `aiogram` library in Python. It is responsible for all customer interactions, including displaying the product catalog, managing the shopping cart, and guiding users through the checkout process. It operates in a stateless manner regarding business logic, delegating all data-related operations to the Backend API. It communicates with Telegram’s servers via webhooks for real-time, push-based updates.
- **Backend API (Core Logic):** The central component of the system, developed using the **FastAPI** framework. It exposes a RESTful API that serves as the single source of truth and business logic. Its responsibilities include:
  - Processing requests from the Telegram Bot and the Admin Panel.
  - Managing all CRUD operations for products, orders, and users.
  - Handling business logic such as inventory checks and order processing workflows.
  - Authenticating and authorizing requests from the Admin Panel using JWT.
  - Interfacing with the PostgreSQL database for data persistence.
  - Communicating with external services through dedicated integration adapters.
- **PostgreSQL Database (Data Persistence):** A robust, open-source relational database that serves as the primary data store. It will house all critical application data, including tables for users, products, product variants, categories, orders, and cart information. The schema is designed to support complex relationships, such as the one-to-many relationship between products and their variants, and to ensure data integrity through constraints and transactions.
- **Admin Panel (Management Interface):** A web-based Single-Page Application (SPA) built with **React**. This interface is exclusively for store administrators. It provides a user-friendly GUI for managing the store’s operations. All data displayed in the admin panel is fetched from the Backend API, and all management actions (e.g., updating an order status, adding a product) are executed by making authenticated API calls.
- **Integration Adapters (External Services):** These are modules within the Backend API responsible for abstracting communication with third-party services. For the MVP, this includes:
  - **Google Sheets Adapter:** Handles authentication and communication with the Google Sheets API for importing product data and exporting order reports.
  - **Analytics Adapter:** Formats and sends e-commerce event data (e.g., `add_to_cart`, `purchase`) to the APIs of Google Analytics and Yandex.Metrika.

3.2. Technology Stack

- **Backend:** Python 3.11+, FastAPI, Pydantic, SQLAlchemy (for ORM), `python-jose` (for JWT).

- **Telegram Bot:** Python 3.11+, aiogram 3.
- **Database:** PostgreSQL 15+.
- **Admin Panel:** JavaScript, React, Axios (for API communication).
- **Deployment:** Docker, Gunicorn/Uvicorn (for serving the FastAPI app).
- **Key Python Libraries:** `google-api-python-client` (for Google Sheets), `openpyxl` and `pandas` (for CSV/XLSX handling), `python-dotenv` (for environment variables).

3.3. Component Interaction Diagram

The following describes the flow of information between the system's components, as would be depicted in a component diagram.

A **User** interacts exclusively with the **Telegram Bot**. The bot, running on a server, receives updates from Telegram's API (ideally via webhooks). When a user sends a command or presses an inline button, the **Telegram Bot** application processes the input. For any action requiring data or business logic (e.g., fetching products, placing an order), the bot makes an HTTP request to the **Backend API**.

The **Backend API** is the central hub. It receives requests from two sources: the **Telegram Bot** and the **Admin Panel**. It processes these requests, applying business rules and interacting with the **PostgreSQL Database** to read or write data. For example, when a user requests to see a product catalog, the bot asks the API, which then queries the database for the relevant products and returns them to the bot for display.

The **Admin Panel**, a React application running in an administrator's web browser, also communicates exclusively with the **Backend API**. When an admin logs in, the panel sends credentials to an authentication endpoint on the API. Upon success, it receives a JWT, which is included in all subsequent requests to protected endpoints for managing products or orders.

The **Backend API** also contains **Integration Adapters**. When an admin triggers an order export to Google Sheets, the API's **Google Sheets Adapter** is invoked to handle the communication. Similarly, when a customer completes a purchase in the bot, the API's **Analytics Adapter** is triggered to send a `purchase` event to both Google Analytics and Yandex.Metrika. This architecture ensures that all external communication is centralized and managed by the backend.

3.4. Architectural Principles

- **Separation of Concerns:** The project is structured by module functionality (domain-driven). For instance, all logic, schemas, and database models related to "orders" are grouped together. This is a scalable approach for a monolithic application, as it keeps related components cohesive and simplifies maintenance. The `src/` directory will contain packages for `auth`, `products`, `orders`, etc., each with its own `router.py`, `service.py`, `schemas.py`, and `models.py`.
- **Dependency Injection:** FastAPI's dependency injection system will be heavily utilized. This pattern is crucial for managing reusable logic such as database sessions (`get_db`), authentication checks (`get_current_admin_user`), and service initializations. Dependencies can be applied at the router level to protect entire modules or at the path operation level for granular control.
- **Asynchronous Operations:** To ensure high performance and responsiveness, the system will leverage `async` and `await` wherever possible. FastAPI is built for async I/O, making it ideal for handling concurrent requests from the bot and admin panel. All database queries and external API calls will be performed asynchronously to prevent blocking the event loop.
- **Webhook-driven Bot:** The Telegram bot will use webhooks instead of polling (`getUpdates`). When an event occurs, Telegram will send an HTTP POST request with an `Update` object to a ded-

icated endpoint on our backend. This is more efficient, scalable, and provides real-time responsiveness, reducing server load and latency.

4. Data Models

The data persistence layer will be managed by a PostgreSQL database. The schema is designed to be normalized and flexible, accommodating a detailed product catalog with variants, a comprehensive order management system, and user information. SQLAlchemy will be used as the ORM to map these tables to Python objects.

4.1. Database Schema

The following tables form the core of the database schema.

`users` Table
Stores information about customers who interact with the bot, primarily captured during the checkout process.
* `id` (BIGINT, Primary Key): Unique identifier for the user, corresponding to their Telegram User ID.
* `full_name` (VARCHAR): The full name of the user.
* `username` (VARCHAR, Nullable): The user's Telegram username.
* `phone_number` (VARCHAR, Nullable): The user's contact phone number.
* `created_at` (TIMESTAMP WITH TIME ZONE): Timestamp of the user's first interaction/order.

`admins` Table
Stores credentials for administrators who can access the web panel.
* `id` (SERIAL, Primary Key): Unique identifier for the admin.
* `username` (VARCHAR, Unique): The login username for the admin.
* `hashed_password` (VARCHAR): The securely hashed password.
* `role` (VARCHAR): Role of the admin (e.g., 'superadmin', 'manager').

`categories` Table
Represents product categories, allowing for a hierarchical structure.
* `id` (SERIAL, Primary Key): Unique identifier for the category.
* `name` (VARCHAR): The name of the category (e.g., "Clothing", "Electronics").
* `parent_id` (INTEGER, Foreign Key to `categories.id`, Nullable): Self-referencing key to create a parent-child hierarchy.

`products` Table
Represents a base product, which acts as a container for its variants.
* `id` (SERIAL, Primary Key): Unique identifier for the product.
* `name` (VARCHAR): The name of the base product (e.g., "Classic T-Shirt").
* `description` (TEXT): A detailed description of the product.
* `category_id` (INTEGER, Foreign Key to `categories.id`): Links the product to a category.
* `is_active` (BOOLEAN, Default: true): A flag to enable or disable the product in the store.

`product_variants` Table
Represents a specific, purchasable version of a product.
* `id` (SERIAL, Primary Key): Unique identifier for the variant.
* `product_id` (INTEGER, Foreign Key to `products.id`): Links the variant to its base product.
* `sku` (VARCHAR, Unique): A unique Stock Keeping Unit for inventory tracking.
* `price` (DECIMAL(10, 2)): The selling price of this specific variant.
* `stock_quantity` (INTEGER): The current number of items in stock for this variant.

`option_types` Table
Defines the types of options a product can have (e.g., “Color”, “Size”).
* `id` (SERIAL, Primary Key): Unique identifier for the option type.
* `name` (VARCHAR, Unique): The name of the option type.

`option_values` Table
Defines the possible values for each option type (e.g., “Red”, “Large”).
* `id` (SERIAL, Primary Key): Unique identifier for the option value.
* `option_type_id` (INTEGER, Foreign Key to `option_types.id`): Links the value to its type.
* `value` (VARCHAR): The actual value (e.g., “M”, “Blue”, “64GB”).

`variant_values` (Junction Table)
A many-to-many relationship linking a product variant to its specific option values. This defines what a variant is (e.g., Variant #101 is defined by the values “Red” and “Large”).
* `variant_id` (INTEGER, Foreign Key to `product_variants.id`): Part of the composite primary key.
* `option_value_id` (INTEGER, Foreign Key to `option_values.id`): Part of the composite primary key.

`orders` Table
Represents a customer’s order.
* `id` (SERIAL, Primary Key): Unique identifier for the order.
* `user_id` (BIGINT, Foreign Key to `users.id`): The customer who placed the order.
* `status` (VARCHAR): The current status of the order (e.g., ‘pending’, ‘processing’, ‘shipped’, ‘delivered’, ‘cancelled’).
* `total_amount` (DECIMAL(10, 2)): The total cost of the order.
* `shipping_address` (TEXT): The full delivery address provided by the user.
* `customer_contact` (TEXT): Contact details provided at checkout (name and phone).
* `created_at` (TIMESTAMP WITH TIME ZONE): Timestamp when the order was placed.
* `updated_at` (TIMESTAMP WITH TIME ZONE): Timestamp of the last status update.

`order_items` Table
A junction table detailing the items within each order.
* `id` (SERIAL, Primary Key): Unique identifier for the order item line.
* `order_id` (INTEGER, Foreign Key to `orders.id`): The order this item belongs to.
* `variant_id` (INTEGER, Foreign Key to `product_variants.id`): The specific product variant ordered.
* `quantity` (INTEGER): The number of units of this variant ordered.
* `price_at_purchase` (DECIMAL(10, 2)): The unit price of the variant at the time of the order, to preserve historical accuracy.

4.2. Relationships and Examples

- **Product-Variant Relationship:** A `product` (e.g., “iPhone 15”) has a one-to-many relationship with `product_variants`. Each `product_variant` represents a purchasable item (e.g., “iPhone 15, 128GB, Blue”).
- **Variant Definition:** A `product_variant` is defined by its association with one or more `option_values` through the `variant_values` table. For example, a specific T-shirt variant would be linked to an `option_value` for “Color” (e.g., ‘Red’) and another for “Size” (e.g., ‘L’).

Example Data Representation (JSON-like):

Consider a product “Classic T-Shirt” with variants.

Product:

```
{
  "id": 1,
  "name": "Classic T-Shirt",
  "description": "A comfortable 100% cotton t-shirt.",
  "category_id": 1
}
```

Option Types and Values:

```
// option_types
[{"id": 1, "name": "Color"}, {"id": 2, "name": "Size"}]

// option_values
[
  {"id": 1, "option_type_id": 1, "value": "Red"},
  {"id": 2, "option_type_id": 1, "value": "Blue"},
  {"id": 3, "option_type_id": 2, "value": "M"},
  {"id": 4, "option_type_id": 2, "value": "L"}
]
```

Product Variants:

```
// product_variants
[
  {
    "id": 101,
    "product_id": 1,
    "sku": "TS-RED-M",
    "price": "19.99",
    "stock_quantity": 50
  },
  {
    "id": 102,
    "product_id": 1,
    "sku": "TS-BLUE-L",
    "price": "21.99",
    "stock_quantity": 30
  }
]
```

Variant Definition (variant_values table):

```
// variant_values
[
  {"variant_id": 101, "option_value_id": 1}, // Links variant 101 to "Red"
  {"variant_id": 101, "option_value_id": 3}, // Links variant 101 to "M"
  {"variant_id": 102, "option_value_id": 2}, // Links variant 102 to "Blue"
  {"variant_id": 102, "option_value_id": 4}  // Links variant 102 to "L"
]
```

This structure allows for immense flexibility in defining products and ensures that inventory and pricing are managed at the most granular level (the specific variant).

5. API Contracts

The Backend API, built with FastAPI, will expose a series of RESTful endpoints. All data exchange will use JSON. Pydantic models will be used for request and response validation, ensuring data integrity.

5.1. Authentication Endpoints

These endpoints are used by the Admin Panel.

`POST /api/v1/auth/token`
* **Description:** Authenticates an administrator and returns JWT access and refresh tokens.
* **Request Body (form data):**
`username: str`
`password: str`
* **Response Body (200 OK):**
`json`
`{`
`"access_token": "string (jwt)",`
`"token_type": "bearer"`
`}`
Note: The refresh token will be set in an `HttpOnly` cookie.
* **Status Codes:** `200 OK`, `401 Unauthorized`.

`POST /api/v1/auth/refresh`
* **Description:** Issues a new access token using a valid refresh token.
* **Request:** Requires the refresh token from the `HttpOnly` cookie.
* **Response Body (200 OK):**
`json`
`{`
`"access_token": "string (jwt)",`
`"token_type": "bearer"`
`}`
* **Status Codes:** `200 OK`, `401 Unauthorized`.

5.2. Public Endpoints (for Telegram Bot)

These endpoints do not require authentication and are consumed by the bot.

`GET /api/v1/products/`
* **Description:** Retrieves a paginated list of active products. Can be filtered by category.
* **Query Parameters:** `category_id: int` (optional), `page: int` (default 1), `size: int` (default 10).
* **Response Body (200 OK):**
`json`
`{`
`"items": [`
`{`
`"id": 1,`
`"name": "Classic T-Shirt",`
`"description": "A comfortable 100% cotton t-shirt."`
`}`
`],`
`"total": 1,`
`"page": 1,`

```
      "size": 10
    }
```

* **Status Codes:** `200 OK`.

`GET /api/v1/products/{product_id}`

* **Description:** Retrieves detailed information for a single product, including its available variants and options.
* **Response Body (200 OK):**

```
json
    {
      "id": 1,
      "name": "Classic T-Shirt",
      "description": "A comfortable 100% cotton t-shirt.",
      "options": [
        {"name": "Color", "values": ["Red", "Blue"]},
        {"name": "Size", "values": ["M", "L"]}
      ],
      "variants": [
        {
          "id": 101,
          "sku": "TS-RED-M",
          "price": "19.99",
          "stock_quantity": 50,
          "attributes": [{"type": "Color", "value": "Red"}, {"type": "Size", "value": "M"}]
        }
      ]
    }
```

* **Status Codes:** `200 OK`, `404 Not Found`.

`POST /api/v1/orders/`

* **Description:** Creates a new order. This is the final step of the checkout process.
* **Request Body:**

```
json
    {
      "user_id": 123456789,
      "user_details": {
        "full_name": "John Doe",
        "phone_number": "+15551234567"
      },
      "shipping_address": "123 Main St, Anytown, USA",
      "items": [
        {"variant_id": 101, "quantity": 2},
        {"variant_id": 102, "quantity": 1}
      ]
    }
```

* **Response Body (201 Created):**

```
json
    {
      "id": 56,
      "status": "pending",
```

```
      "total_amount": "61.97",
      "items": [
        {"product_name": "Classic T-Shirt (Red, M)", "quantity": 2},
        {"product_name": "Classic T-Shirt (Blue, L)", "quantity": 1}
      ]
    }
```

* **Status Codes:** `201 Created`, `400 Bad Request` (e.g., item out of stock), `422 Unprocessable Entity`.

5.3. Admin Endpoints (for Admin Panel)

These endpoints require a valid JWT access token in the `Authorization` header.

`GET /api/v1/admin/orders/`

* **Description:** Retrieves a paginated list of all orders. Can be filtered by status.
* **Query Parameters:** `status: str` (optional), `page: int`, `size: int`.
* **Response Body (200 OK):**

```
json
    {
      "items": [
        {
          "id": 56,
          "user_id": 123456789,
          "status": "pending",
          "total_amount": "61.97",
          "created_at": "2025-11-30T10:00:00Z"
        }
      ],
      "total": 1,
      "page": 1,
      "size": 20
    }
```

* **Status Codes:** `200 OK`, `401 Unauthorized`.

`GET /api/v1/admin/orders/{order_id}`

* **Description:** Retrieves full details for a specific order.
* **Response Body (200 OK):**

```
json
    {
      "id": 56,
      "user_id": 123456789,
      "status": "pending",
      "total_amount": "61.97",
      "shipping_address": "123 Main St, Anytown, USA",
      "customer_contact": "John Doe, +15551234567",
      "created_at": "2025-11-30T10:00:00Z",
      "items": [
        {
          "variant_id": 101,
          "sku": "TS-RED-M",
          "product_name": "Classic T-Shirt",
```

```
          "quantity": 2,
          "price_at_purchase": "19.99"
        }
      ]
    }
```

* **Status Codes:** `200 OK`, `401 Unauthorized`, `404 Not Found`.

`PATCH /api/v1/admin/orders/{order_id}`
* **Description:** Updates the status of an order.
* **Request Body:**

```
json
    {
      "status": "processing"
    }
```

* **Response Body (200 OK):** The updated order object.
* **Status Codes:** `200 OK`, `400 Bad Request`, `401 Unauthorized`, `404 Not Found`.

`POST /api/v1/admin/products/`
* **Description:** Creates a new product with its variants.
* **Request Body:** A complex object representing the product and its variants, similar to the GET response but without IDs.
* **Response Body (201 Created):** The newly created product object with generated IDs.
* **Status Codes:** `201 Created`, `401 Unauthorized`, `422 Unprocessable Entity`.

`PUT /api/v1/admin/products/{product_id}`
* **Description:** Updates an existing product and its variants.
* **Request Body:** The full product object to be updated.
* **Response Body (200 OK):** The updated product object.
* **Status Codes:** `200 OK`, `401 Unauthorized`, `404 Not Found`, `422 Unprocessable Entity`.

`DELETE /api/v1/admin/products/{product_id}`
* **Description:** Deletes a product and all its associated variants.
* **Response Body (204 No Content):** Empty response.
* **Status Codes:** `204 No Content`, `401 Unauthorized`, `404 Not Found`.

6. Telegram Bot Flows

The user experience within the Telegram bot will be guided by a series of interactive flows managed by `aiogram`'s Finite State Machine (FSM) and inline keyboards. The design prioritizes simplicity and clarity.

6.1. User Scenarios

Scenario 1: First-time User and Catalog Browsing
1. **User:** Sends `/start` to the bot.
2. **Bot:** Responds with a welcome message and a main menu inline keyboard: `[Catalog]`, `[My Cart]`, `[Help]`.
3. **User:** Clicks `[Catalog]`.
4. **Bot:** Fetches top-level categories from the API (`GET /api/v1/categories/`) and displays them as an inline keyboard.
5. **User:** Clicks a category, e.g., `[Clothing]`.
6. **Bot:** Fetches the first page of products in that category from the API (`GET /api/v1/products/?cat-`

`egory_id=X`) and displays them as a list in a message, with an inline keyboard for pagination (`<<` , `<` , `Page 1/5` , `>` , `>>`) and a button for each product. The `aiogram-inline-paginations` library will be used to generate this keyboard.
7. **User:** Clicks a product, e.g., `[Classic T-Shirt]` .
8. **Bot:** Fetches detailed product info (`GET /api/v1/products/{id}`) and displays the description, price range, and an inline keyboard for selecting variants.

Scenario 2: Selecting a Product with Variants and Adding to Cart
1. **Context:** User is viewing the “Classic T-Shirt” product details.
2. **Bot:** Displays an inline keyboard with the first option type, e.g., `[Select Color]` .
3. **User:** Clicks `[Select Color]` .
4. **Bot:** Edits the message to show available colors: `[Red]` , `[Blue]` , `[Back]` .
5. **User:** Clicks `[Red]` .
6. **Bot:** Records “Red” in the FSM context. Edits the message to show the next option type: `[Select Size]` .
7. **User:** Clicks `[Select Size]` .
8. **Bot:** Edits the message to show available sizes for the red color: `[M]` , `[L]` , `[Back]` .
9. **User:** Clicks `[M]` .
10. **Bot:** Records “M” in the FSM context. Now that all options are selected, the bot identifies the specific variant (`TS-RED-M`). It displays the final variant details (price, stock) and an inline keyboard: `[Add to Cart]` , `[Back to Product]` .
11. **User:** Clicks `[Add to Cart]` .
12. **Bot:** Stores the selected variant ID and quantity in the user’s session (managed by the FSM storage, e.g., Redis). Responds with a confirmation: “✅ ‘Classic T-Shirt (Red, M)’ added to your cart.” and shows the main menu again.

Scenario 3: Checkout Process
1. **User:** Clicks `[My Cart]` from the main menu.
2. **Bot:** Retrieves the cart contents from the FSM storage. Displays a summary of items, quantities, and total price. Provides an inline keyboard: `[Checkout]` , `[Clear Cart]` , `[Back to Menu]` .
3. **User:** Clicks `[Checkout]` .
4. **Bot:** Enters the ‘checkout’ state using the FSM. It asks the first question: “Please enter your full name.”
5. **User:** Sends their name as a message.
6. **Bot:** Validates and stores the name in the FSM context. Asks the next question: “Please provide your contact phone number.”
7. **User:** Sends their phone number.
8. **Bot:** Validates and stores the phone number. Asks the final question: “Please enter your full delivery address.”
9. **User:** Sends their address.
10. **Bot:** Stores the address. Displays a final confirmation message with all order details (items, total price, name, phone, address) and an inline keyboard: `[Confirm Order]` , `[Cancel]` .
11. **User:** Clicks `[Confirm Order]` .
12. **Bot:** Sends all collected data to the backend API (`POST /api/v1/orders/`).
13. **API:** Processes the order, creates records in the database, and decrements stock.
14. **Bot:** Upon a successful API response, it clears the user’s cart state and sends a final message: “🎉 Your order #56 has been placed successfully! We will notify you about status updates.” It also sends a notification to the admin group/channel.

6.2. State Machine (FSM)

`aiogram`'s FSM is critical for managing multi-step interactions. We will define state groups for different processes.

- **`ProductSelectionStates`:**
  - `choosing_option`: A generic state for when a user is cycling through variant options (color, size, etc.). The context data will store which option is next and what has been selected so far.
- **`CheckoutStates`:**
  - `waiting_for_name`: The bot is waiting for the user to enter their full name.
  - `waiting_for_phone`: The bot is waiting for the user's phone number.
  - `waiting_for_address`: The bot is waiting for the delivery address.
  - `confirmation`: The bot is waiting for the user to confirm the final order details.

The `aiogram-dialog` library will be considered to simplify the management of these states and their corresponding UI (windows, widgets, keyboards). It provides a higher-level abstraction over the raw FSM, reducing boilerplate code for message display and handler logic, making the conversation flows more declarative and easier to maintain.

6.3. Admin Notifications

The bot will also serve as a notification tool for administrators. A dedicated, private Telegram channel or group will be used for this purpose.

- **New Order Notification:** When an order is successfully placed via `POST /api/v1/orders/`, the backend will trigger a bot message to the admin channel. The message will be richly formatted and contain all essential order details:
  - Order ID
  - Customer Name and Contact Info
  - Shipping Address
  - List of Items (Product, Variant, Quantity)
  - Total Amount
  - A direct link to the order page in the admin panel (e.g., `https://admin.yourstore.com/orders/56`).
- **Low Stock Alert:** A background job on the backend can periodically check `product_variants.stock_quantity`. If a variant's stock falls below a predefined threshold, a notification will be sent to the admin channel, prompting them to restock.

7. Integrations

The system will integrate with several external services to enhance functionality for administrators and provide business intelligence.

7.1. Google Sheets

This integration allows for bulk management of the product catalog and easy exportation of order data for analysis.

- **Authentication:** The backend will use a **Service Account** for authentication with the Google Sheets API. This is suitable for server-to-server interactions where no user consent is required. A

JSON key file for the service account will be generated from the Google Cloud Console and stored securely as a secret. The service account's email address must be granted editor access to the target Google Sheet.

- **Product Import:**
  1. An administrator prepares a Google Sheet with a predefined structure (columns for `sku`, `product_name`, `description`, `price`, `stock_quantity`, `option_color`, `option_size`, etc.).
  2. In the admin panel, the admin provides the URL of the sheet and triggers an import.
  3. The backend API receives the request. The Google Sheets Adapter uses the `google-api-python-client` library to fetch all data from the specified sheet.
  4. The data is parsed, validated, and transformed into the application's internal data structures.
  5. The backend then performs a batch update/create operation on the `products` and `product_variants` tables in the PostgreSQL database.
- **Order Export:**
  1. An administrator selects a date range in the admin panel and clicks "Export to Google Sheets."
  2. The backend API queries the database for all orders within that range.
  3. The Google Sheets Adapter formats the order data into rows and columns.
  4. It uses the `spreadsheets().values().batchUpdate()` method to write the data to a new or existing Google Sheet in a single, efficient API call. This reduces network overhead and is faster than writing row by row.

7.2. CSV / XLSX Import and Export

This functionality provides an alternative to Google Sheets for data management directly through the admin panel.

- **File Handling:** The backend will use the `pandas` and `openpyxl` libraries to handle these file formats.
- **Import Flow:**
  1. An admin uploads a CSV or XLSX file through a form in the React admin panel.
  2. The file is sent to a dedicated endpoint on the FastAPI backend.
  3. FastAPI receives the `UploadFile`. The backend uses `pandas.read_csv()` or `pandas.read_excel()` to load the file content into a DataFrame.
  4. The DataFrame is then iterated over, with each row being validated against Pydantic models before being inserted or updated in the database. This process will be run as a background task to avoid blocking the HTTP response for large files.
- **Export Flow:**
  1. An admin requests an export of orders or products.
  2. The backend queries the database and loads the data into a pandas DataFrame.
  3. The DataFrame is converted to the desired format using `df.to_csv()` or `df.to_excel()`.
  4. FastAPI returns the file as a `StreamingResponse` or `FileResponse`, allowing the admin's browser to download it.

7.3. Google Analytics and Yandex.Metrika

To track user behavior and e-commerce performance, the system will send events to both GA4 and Yandex.Metrika. Since the user interaction happens within Telegram, tracking cannot be done with client-side JavaScript. Instead, server-to-server events will be sent from the backend.

- **Event Tracking:** The backend API will be responsible for sending events at key moments in the user journey.
  - `view_item`: When a user views a product's details in the bot.
  - `add_to_cart`: When a user adds an item to their cart.
  - `begin_checkout`: When a user starts the checkout process.
  - `purchase`: When a user successfully confirms an order.
- **Google Analytics (GA4) Integration:**
  - **Method:** Events will be sent using the **Google Analytics Measurement Protocol API**. This is a REST API that allows server-side hits to be sent directly to GA4 servers.
  - **Implementation:** The Analytics Adapter will construct an HTTP POST request to the GA4 endpoint. The request body will be a JSON payload containing the `client_id` (a persistent, anonymous identifier for the Telegram user), the event name (e.g., `purchase`), and e-commerce parameters (e.g., `transaction_id`, `value`, `currency`, `items` array).
- **Yandex.Metrika Integration:**
  - **Method:** Similar to GA, events will be sent server-side. Yandex.Metrika supports e-commerce data collection via a JavaScript API and a `dataLayer` object. For server-to-server tracking, we will simulate this by sending data to a special endpoint or using a library that wraps this functionality.
  - **Implementation:** The Analytics Adapter will be configured with the Yandex.Metrika tag ID. When a `purchase` event occurs, it will construct a payload that mirrors the structure expected by Yandex's e-commerce module (containing details about the purchase and the products) and send it via an HTTP request.
- **Benefits of Dual Tracking:** Using both platforms provides data redundancy and allows for cross-verification. Yandex.Metrika offers features like Session Replay (though less applicable to a bot context) and is popular in the CIS region, while Google Analytics is a global standard. Differences in their data models (e.g., how they define a "bounce" or count pageviews) can provide a more nuanced understanding of user engagement.

8. Security

Security is a paramount concern and will be addressed at multiple levels of the application stack.

8.1. Authentication and Authorization

- **Admin Panel Authentication:** Access to the admin panel and its corresponding API endpoints will be protected by a robust JWT-based authentication system.
  1. **Login:** An admin submits their username and password. The backend verifies the credentials against the `admins` table (with passwords hashed using a strong algorithm like bcrypt via `passlib`).
  2. **Token Issuance:** Upon successful login, the server generates two tokens: a short-lived **access token** (e.g., 15-30 minutes) and a long-lived **refresh token** (e.g., 7 days).
  3. **Token Storage:** The access token is sent in the response body and stored in memory (e.g., React state) on the client-side. It should **not** be stored in `localStorage` to prevent XSS at-

tacks. The refresh token is sent as a secure, `HttpOnly` cookie, making it inaccessible to client-side JavaScript.

4. **API Requests:** The React client includes the access token in the `Authorization: Bearer <token>` header for all requests to protected endpoints.
5. **Token Refresh:** When the access token expires, the client will receive a `401 Unauthorized` status. An Axios interceptor will then automatically make a request to the `/api/v1/auth/refresh` endpoint. This request will include the refresh token from the cookie. If the refresh token is valid, the server issues a new access token, and the original failed request is retried seamlessly.

- **Telegram Bot:** The bot communicates with our public API endpoints, which do not require authentication. However, the webhook endpoint itself must be secured. A secret token will be passed in the `setWebhook` call, and the backend will verify this token on every incoming request from Telegram to ensure it originates from a legitimate source.

8.2. Secrets Management

All sensitive information, such as the Telegram bot token, database connection string, JWT secret key, and Google API credentials, will be managed securely.
* **Development:** In the local development environment, secrets will be stored in a `.env` file. This file will be loaded into the application's environment at runtime using the `python-dotenv` library. The `.env` file will be explicitly listed in `.gitignore` to prevent it from ever being committed to version control.
* **Production:** In production, secrets will be managed using a dedicated secrets management service, such as **AWS Secrets Manager**, **Google Cloud Secret Manager**, or **HashiCorp Vault**. The application will be granted IAM permissions to fetch these secrets at startup. This approach provides centralized management, auditing, and easy rotation of secrets without code changes or deployments. Environment variables injected by the hosting platform (e.g., Heroku Config Vars, Docker environment variables) are also a viable and common alternative.

8.3. Input Validation

All incoming data from external sources (user input in the bot, API requests from the admin panel) will be rigorously validated.
* **FastAPI and Pydantic:** FastAPI's use of Pydantic models for request bodies provides automatic, out-of-the-box data validation. If an incoming request does not conform to the defined schema (e.g., wrong data type, missing required field), FastAPI will automatically return a `422 Unprocessable Entity` response with a detailed error message. This prevents malformed or malicious data from reaching the application's business logic.
* **Bot Input:** For multi-step FSM flows in the bot (like checkout), each piece of user input (name, phone number, address) will be validated using regular expressions or other checks before being stored in the state context and moving to the next step.

8.4. Compliance with 152-ФЗ

To comply with the Russian Federal Law "On Personal Data" (152-ФЗ), the following measures will be implemented:
* **Data Localization:** All servers used for the application, including the web server hosting the FastAPI application and the PostgreSQL database server, will be physically located within the Russian Federation. This is a strict requirement for storing the personal data of Russian citizens.
* **User Consent:** During the checkout process, before collecting any personal data (name, phone, address), the bot will present the user with a link to the Privacy Policy and Terms of Service and require them to explicitly agree to the processing of their personal data by clicking an "I Agree" button. This

consent will be logged.
* **Privacy Policy:** A clear and accessible privacy policy will be drafted, detailing what personal data is collected, for what purpose, how it is stored and protected, and for how long. A link to this policy will be available from the bot's main menu or help section.
* **Data Security:** All technical measures described in this security section (encryption, access control, secure secrets management) contribute to fulfilling the law's requirement to protect personal data from unauthorized access.

9. Performance and Scalability

The architecture is designed to provide good performance for the MVP launch and to be scalable as the user base and product catalog grow.

9.1. Server Requirements (MVP)

- **Application Server (FastAPI + aiogram):** A virtual private server (VPS) with at least **2 vCPUs** and **4 GB of RAM**. This should be sufficient to handle the asynchronous workload of the FastAPI application and the bot logic for a moderate number of concurrent users.
- **Database Server (PostgreSQL):** A separate VPS or a managed database service with **2 vCPUs**, **4 GB of RAM**, and fast SSD storage. Separating the database from the application server prevents resource contention.
- **Admin Panel:** As a static React application, it can be hosted on a simple object storage service (like AWS S3 with CloudFront) or on the same server as the backend, served by Nginx.

9.2. Optimization Strategies

- **Asynchronous Backend:** The core performance feature is the use of FastAPI's `async` capabilities. All I/O-bound operations, including database queries (using an async driver like `asyncpg` with SQLAlchemy) and calls to external APIs (Google, Telegram, Analytics), will be non-blocking. This allows a single server process to handle many concurrent connections efficiently.
- **Database Optimization:**
  - **Indexing:** Proper indexes will be created on foreign key columns (`product_id` , `category_id` , `order_id` , etc.) and frequently queried columns (e.g., `orders.status` , `products.name`). This will dramatically speed up read operations.
  - **Query Optimization:** Complex queries will be analyzed using `EXPLAIN ANALYZE` to ensure they are using efficient execution plans. N+1 query problems will be avoided by using SQLAlchemy's eager loading options (`selectinload`) where appropriate.
- **Efficient Bot Communication:** Using **webhooks** instead of long polling is a fundamental performance choice. It eliminates unnecessary HTTP requests from our server to Telegram and provides instant updates, reducing both latency and server load.
- **Caching:** For frequently accessed and rarely changing data, such as the list of product categories, a caching layer (e.g., Redis) can be introduced. This would reduce the number of queries to the database. This is considered a post-MVP optimization.
- **Background Tasks:** For long-running tasks initiated from the admin panel, such as importing a large XLSX file or generating a complex report, FastAPI's `BackgroundTasks` will be used. This allows the API to immediately return a response to the client while the task runs in the background, preventing HTTP timeouts and improving the user experience.

9.3. Scalability Path

- **Horizontal Scaling:** The backend API is designed to be stateless. All session-related data (e.g., user's shopping cart) is managed by the `aiogram` FSM, which can use a distributed backend like

Redis. This means we can scale the application horizontally by simply adding more instances of the application server behind a load balancer.

- **Database Scaling:** If the database becomes a bottleneck, it can be scaled vertically (upgrading to a more powerful server) or horizontally through read replicas. A read replica can handle the read-heavy load from catalog browsing, while the primary database handles writes (orders, updates).
- **CDN for Media:** Product images and other static media associated with the store should be served through a Content Delivery Network (CDN) to reduce latency for users and offload traffic from the main application server.

10. Roadmap for Future Phases

The MVP establishes a solid foundation. The following features are proposed for subsequent development phases to enhance functionality, improve user engagement, and expand business capabilities.

10.1. Phase 2: Enhancing Core Functionality

- **Payment Gateway Integration:** Integrate with a payment provider (e.g., Stripe, YooKassa) to allow users to pay for their orders directly within the checkout flow using credit cards or other online payment methods. This would involve a web app integration within Telegram.
- **Customer Accounts and Order History:** Allow users to create an account linked to their Telegram ID. Authenticated users could view their order history, track shipment status directly in the bot, and save their shipping addresses for faster checkout.
- **Advanced Admin Dashboard:** Enhance the admin panel with a more comprehensive dashboard, including visual analytics, sales charts (daily, weekly, monthly), top-selling products, and customer lifetime value metrics.
- **Promotions and Discount Codes:** Implement a system for creating and managing discount codes. Users could apply a code during checkout to receive a percentage or fixed amount off their order.
- **Product Reviews and Ratings:** Allow customers to leave reviews and ratings for products they have purchased. These reviews would be displayed on the product detail page in the bot.

10.2. Phase 3: Scaling and Automation

- **Multi-Language and Multi-Currency Support:** Internationalize the bot and admin panel interfaces. Allow administrators to provide product descriptions in multiple languages and enable customers to view prices and check out in different currencies.
- **Integration with Delivery Services:** Integrate with the APIs of shipping carriers (e.g., CDEK, Boxberry) to automatically calculate shipping costs based on the user’s address, generate shipping labels, and provide real-time tracking information.
- **Marketing and CRM Integration:**
  - **Abandoned Cart Recovery:** Implement a system to automatically send follow-up messages to users who start a checkout process but do not complete it.
  - **Push Notifications:** Enable administrators to send targeted promotional messages or product announcements to segments of the user base (e.g., users who have purchased from a specific category).
  - **CRM Sync:** Set up a two-way synchronization with a CRM system (e.g., Kommo, Bitrix24) to manage customer relationships and sales funnels more effectively.

- **Advanced Inventory Management:** Introduce features for managing multiple warehouses, setting up stock alerts with more complex rules, and tracking inventory movements between locations.
- **Recommendation Engine:** Implement a simple recommendation engine to suggest related or frequently bought-together products on product pages or in the cart, aiming to increase the average order value.

References

1. aiogram-inline-paginations - PyPI (https://pypi.org/project/aiogram-inline-paginations/)
2. Friends, hello! - Medium (https://medium.com/@amverait/friends-hello-8460dfe86ef1)
3. daniilshamraev/aiogram-inline-paginations - GitHub (https://github.com/daniilshamraev/aiogram-inline-paginations)
4. aiogram documentation - Read the Docs (https://app.readthedocs.org/projects/aiogram/downloads/pdf/latest/)
5. Problems with pagination callbacks in aiogram3 - Stack Overflow (https://stackoverflow.com/questions/77256562/problems-with-pagination-callbacks-in-aiogram3)
6. zhanymkanov/fastapi-best-practices - GitHub (https://github.com/zhanymkanov/fastapi-best-practices)
7. Bigger Applications - FastAPI (https://fastapi.tiangolo.com/tutorial/bigger-applications/)
8. How to structure your FastAPI projects - Medium (https://medium.com/@amirm.lavasani/how-to-structure-your-fastapi-projects-0219a6600a8f)
9. How to Implement Dependency Injection in FastAPI - freeCodeCamp (https://www.freecodecamp.org/news/how-to-implement-dependency-injection-in-fastapi/)
10. FastAPI Architecture - GeeksforGeeks (https://www.geeksforgeeks.org/python/fastapi-architecture/)
11. e-commerce database design - products with optional variations and scaled prices - Stack Overflow (https://stackoverflow.com/questions/43582303/e-commerce-database-design-products-with-optional-variations-and-scaled-prices)
12. Product DB association for e-commerce - Database Administrators Stack Exchange (https://dba.stackexchange.com/questions/209951/product-db-association-for-e-commerce)
13. Schema design for products with multiple variants/attributes - Database Administrators Stack Exchange (https://dba.stackexchange.com/questions/123467/schema-design-for-products-with-multiple-variants-attributes)
14. kdsuneraavinash/neomerce - GitHub (https://github.com/kdsuneraavinash/neomerce)
15. Best table structure for multiple product variants - SitePoint Community (https://www.sitepoint.com/community/t/best-table-structure-for-multiple-product-variants/37444)
16. Telegram Bot API (https://core.telegram.org/bots/api)
17. Webhooks - Telegram Bots API (https://core.telegram.org/bots/webhooks)
18. Telegram Webhooks: A Comprehensive Guide for 2024 - Hevo Data (https://hevodata.com/learn/telegram-webhooks/)
19. Webhook - grammY (https://gramio.dev/updates/webhook)
20. How to Set up and Test Telegram Bot Webhook - Pinggy.io (https://pinggy.io/blog/how_to_set_up_and_test_telegram_bot_webhook/)

21. Notifications Telegram for WooCommerce – WordPress plugin - WordPress.com (https://wordpress.com/plugins/notifications-telegram-for-woocommerce)
22. Telegram notifications - Kommo (https://cmdf5.com/widgets/telegram_notifications)
23. WooCommerce Telegram Order Notification WordPress Plugin by welaunch - CodeCanyon (https://codecanyon.net/item/woocommerce-telegram-order-notification-wordpress-plugin/46855838)
24. Notification for Telegram – WordPress plugin - WordPress.com (https://wordpress.com/plugins/notification-for-telegram)
25. Telegram for OpenCart - opencartbot.com (https://opencartbot.com/en/telegram)
26. Python Quickstart - Google Sheets API (https://developers.google.com/workspace/sheets/api/quickstart/python)
27. A Step-by-Step Guide to Google Spreadsheet Authentication and Automation with Python - Medium (https://medium.com/@jseid212/a-step-by-step-guide-to-google-spreadsheet-authentication-and-automation-with-python-68512060ab01)
28. Batch operations - Google Sheets API (https://developers.google.com/sheets/api/guides/batch)
29. Read a google sheet with python using API key instead of Oauth client ID - Stack Overflow (https://stackoverflow.com/questions/44899425/read-a-google-sheet-with-python-using-api-key-instead-of-oauth-client-id-python)
30. The Comprehensive Guide to Google Sheets with Python - Understanding Data (https://understandingdata.com/posts/the-comprehensive-guide-to-google-sheets-with-python/)
31. Convert Excel to CSV in Python - GeeksforGeeks (https://www.geeksforgeeks.org/python/convert-excel-to-csv-in-python/)
32. Converting CSV to XLSX, numbers stored as text - Python Help (https://discuss.python.org/t/converting-csv-to-xlsx-numbers-stored-as-text/36527)
33. How to Convert an Excel File to a CSV File in Python - Data to Fish (https://datatofish.com/excel-to-csv-python/)
34. Python convert csv to xlsx - Stack Overflow (https://stackoverflow.com/questions/17684610/python-convert-csv-to-xlsx)
35. How to save an excel worksheet as csv? - Stack Overflow (https://stackoverflow.com/questions/10802417/how-to-save-an-excel-worksheet-as-csv)
36. Refresh Token - FastAPI-JWT-Auth (https://indominusbyte.github.io/fastapi-jwt-auth/usage/refresh/)
37. How to implement refresh tokens? - GitHub Discussions (https://github.com/fastapi-users/fastapi-users/discussions/350)
38. Token-based Authentication with FastAPI - Medium (https://gh0stfrk.medium.com/token-based-authentication-with-fastapi-7d6a22a127bf)
39. Bulletproof JWT Authentication in FastAPI: A Complete Guide - Medium (https://medium.com/@ancilartech/bulletproof-jwt-authentication-in-fastapi-a-complete-guide-2c5602a38b4f)
40. Help with JWT auth flow - Reddit (https://www.reddit.com/r/FastAPI/comments/1h94ahs/help_with_jwt_auth_flow/)
41. Search results for 'keywords:yandex metrica' - npm (https://www.npmjs.com/search?q=keywords:yandex+metrica)
42. E-commerce - Yandex.Metrica Help (https://yandex.com/support/metrica/data/e-commerce.html)
43. Performance - Yandex.Metrica (https://metrica.yandex.com/about/info/performance)
44. Yandex Metrica or Google Analytics? The value of using both - K-Bridge (https://www.kbridge.org/en/yandex-metrica-or-google-analytics-the-value-of-using-both/)
45. Yandex Metrica Bot - DataDome (https://datadome.co/bots/yandex-metrica/)
46. Best practices python where to store api keys/tokens - Stack Overflow (https://stackoverflow.com/questions/56995350/best-practices-python-where-to-store-api-keys-tokens)

47. Secrets & API keys management best practices - GitGuardian Blog (https://blog.gitguardian.com/secrets-api-management/)
48. How should I be storing API keys? - Reddit (https://www.reddit.com/r/learnpython/comments/r0z4ad/how_should_i_be_storing_api_keys/)
49. Securing Sensitive Data in Python: Best Practices for Storing API Keys and Credentials - System Weakness (https://systemweakness.com/securing-sensitive-data-in-python-best-practices-for-storing-api-keys-and-credentials-2bee9ede57ee)
50. What's the safest way to store users' api-keys/secrets? - Reddit (https://www.reddit.com/r/learnpython/comments/1b9pl5s/whats_the_safest_way_to_store_users_apikeyssecrets/)