ChatGPT

Proyecto Web: Seguimiento de Cotizaciones de Criptomonedas

Visión General del Proyecto

Este proyecto consiste en una plataforma web para dar seguimiento a precios de criptomonedas, diseñada con una arquitectura de microservicios en Docker. La plataforma permitirá **agregar criptomonedas** para seguimiento, **obtener su cotización diaria** desde una API pública y almacenar históricos en una base de datos PostgreSQL. Además, ofrecerá un **dashboard web interactivo** para consultar las series de precios almacenadas, agilizando el análisis con DuckDB. Se considera un diseño **moderno, funcional y seguro**, preparado para futuras extensiones (como autenticación de usuarios, más indicadores, etc.). A continuación se detallan los componentes, tecnologías y mejores prácticas para implementar este proyecto.

Arquitectura y Componentes (Docker)

El proyecto se estructurará en contenedores Docker mediante *Docker Compose*, separando responsabilidades en servicios independientes:

- **Base de Datos (PostgreSQL)**: Un contenedor con PostgreSQL que almacenará las criptomonedas registradas y sus precios diarios históricos. La persistencia de datos se puede asegurar montando un volumen Docker para la carpeta de datos de PostgreSQL.
- **Aplicación Web (Flask + Gunicorn)**: Un contenedor para la aplicación web desarrollada en Python (Flask). Este contendrá la lógica de backend (endpoints REST/HTML), la integración con la base de datos, la consulta a APIs de precios y la generación de la interfaz frontend. Flask es adecuado por ser ligero y fácil de usar, ideal para un desarrollo rápido, respaldado por la robustez de Python y su ecosistema [1].
- **Red Interna Segura**: Docker Compose creará una red interna donde la app Flask se comunica con PostgreSQL usando credenciales seguras. El puerto de PostgreSQL *no* se expondrá a internet (solo la aplicación web puede accederlo), reduciendo la superficie de ataque.
- (Opcional) **Tareas Programadas**: Para la actualización diaria de precios, se puede incluir un componente para ejecución de cron (por ejemplo, usando *cron* de Linux en el contenedor Flask o una librería de *scheduler* en Python) que llame automáticamente a la función de actualización cada 24 horas. Alternativamente, se puede confiar en un trigger manual inicial (botón "Actualizar cotizaciones") y luego automatizar con un servicio externo de cron si se prefiere simplicidad en esta versión de desarrollo.

Comunicación: La aplicación Flask expondrá puertos HTTP (por ejemplo, http://localhost:8000) para el frontend web. Para entornos de producción, se recomienda enrutamiento a través de un servidor web (Nginx) que sirva de *reverse proxy* y gestione HTTPS. La base de datos usará la cadena de conexión interna (ejemplo: `postgresql://user:pass@db:5432/mydb`). Las credenciales sensibles y configuraciones (como la URL de la API de cotizaciones, claves API si hubiera, contraseñas DB, etc.) se definirán vía **variables**

de entorno en el `docker-compose.yml` o en ficheros de entorno, nunca hardcodeadas en el código fuente, siguiendo buenas prácticas de seguridad.

Tecnologías y Lenguaje de Programación

Se propone utilizar **Python** como lenguaje principal, dado que cumple con todos los requisitos del proyecto y es la preferencia indicada. Python cuenta con bibliotecas maduras para desarrollo web, acceso a base de datos, llamados HTTP a APIs externas y análisis de datos. En particular, el microframework **Flask** ofrece la simplicidad y flexibilidad necesarias para construir rápidamente la aplicación web [2]. Python es reconocido por su sencillez y legibilidad, facilitando el desarrollo y mantenimiento. Además, su rico ecosistema (por ejemplo, frameworks como Flask, ORMs como SQLAlchemy, bibliotecas científicas) permite un desarrollo rápido de aplicaciones sin sacrificar seguridad [1].

Alternativas: Aunque Python/Flask es la opción recomendada, vale mencionar alternativas: - **Node.js (JavaScript)** con frameworks como Express podría emplearse, existiendo también integración para DuckDB (módulo npm) y drivers para PostgreSQL. Sin embargo, implicaría escribir la lógica en JavaScript/TypeScript y gestionar la integración de duckDB de forma distinta. Dado que Python es preferido y ofrece alta productividad en este contexto, seguiremos con Python. - **Otros**: Lenguajes como **Go** o **Ruby** también pueden conectarse a Postgres y consumir DuckDB (DuckDB ofrece bindings en C/C++ que podrían usarse), pero no proporcionan ventajas claras frente a la sencillez de Python para un prototipo.

En resumen, **Python + Flask** es la mejor elección por su equilibrio entre facilidad, capacidades técnicas y comunidad de soporte. Permitirá también integrar DuckDB de forma directa, ya que DuckDB se integra fuertemente con Python, facilitando la manipulación y análisis de datos de forma rápida [3].

Base de Datos: PostgreSQL

Para el almacenamiento principal de la aplicación se usará **PostgreSQL** en un contenedor Docker dedicado. PostgreSQL brinda confiabilidad, integridad ACID y capacidad de escalar en volumen de datos, más que suficiente dado que el proyecto almacenará esencialmente precios diarios (pocos registros por criptomoneda por año). La estructura inicial de la base de datos podría incluir al menos dos tablas principales:

- **Tabla de Criptomonedas**: con campos como `id` (clave primaria), `nombre` o `símbolo`, y posiblemente identificador usado por la API de cotización (por ejemplo, el ID de CoinGecko para esa moneda). Aquí se registran las criptos que el usuario desea seguir.
- **Tabla de Precios**: con campos `id` (PK), `cripto_id` (FK a Criptomoneda), `fecha` (día de la cotización, almacenada como date), `precio` (valor de cierre o actual del día) y quizás otros datos como volumen o capitalización si se requieren. Esta tabla contendrá un registro por día por criptomoneda seguida.

Se pueden definir las relaciones adecuadas (clave foránea de precios a criptomonedas). También se pueden añadir índices por (`cripto_id`, `fecha`) para acelerar consultas de series temporales.

Almacenamiento Histórico: Cuando se agrega una nueva cripto para seguimiento, la aplicación podría opcionalmente obtener datos históricos iniciales (por ejemplo, precios del último año) para poblar la serie

desde el inicio. Esto puede hacerse llamando a la API de precios con endpoints históricos (CoinGecko provee datos históricos diarios fácilmente). De no hacerse, la base de datos simplemente empezará a llenarse con datos diarios a partir del día actual.

Consulta de Datos: La aplicación web consultará esta base de datos para mostrar listas de criptos registradas, insertar nuevos registros de precio, y obtener la serie histórica al graficar. Se usará un driver como **psycopg2** o un ORM como **SQLAlchemy** para interactuar con Postgres desde Flask. Es importante utilizar **consultas parametrizadas o un ORM** para prevenir inyecciones SQL al construir consultas con datos de usuario [4]. Por ejemplo, al insertar un nuevo símbolo de criptomoneda ingresado por el usuario, usar sentencias preparadas en vez de concatenar SQL manualmente.

Integración de DuckDB para Análisis Rápido

Una pieza clave es el uso de **DuckDB** para agilizar análisis complejos sobre los datos de precios. DuckDB es una base de datos analítica embebida, **columar en memoria**, diseñada para cargas de trabajo analíticas (OLAP) con consultas SQL veloces incluso sobre millones de filas [5]. Su inclusión en el proyecto mejora las capacidades de análisis de dos formas principales:

- **Consultas Analíticas Rápidas**: DuckDB puede usarse dentro de la aplicación Python para ejecutar consultas complejas sobre los datos históricos sin recargar la base de datos Postgres. Por ejemplo, cálculos de indicadores técnicos (medias móviles, RSI, correlaciones entre activos, etc.) se pueden hacer vía SQL en DuckDB, el cual está optimizado para escaneo de columnas y cálculos vectorizados en memoria [5]. Esto evita saturar a PostgreSQL con cálculos pesados; la serie de precios necesaria se puede extraer de Postgres (por ejemplo, como un DataFrame de pandas o directamente mediante el conector de DuckDB a Postgres) y luego consultarse en DuckDB local.
- **Integración Directa con Postgres**: Existe una extensión *pg_scanner* que permite a DuckDB conectarse y consultar datos de una instancia Postgres directamente vía SQL estándar [6]. En este proyecto, una opción sencilla es que la aplicación Flask, al solicitar datos para graficar o analizar, cree una conexión DuckDB, "atache" la base de datos Postgres temporalmente y ejecute una consulta. Esto retornaría un resultado muy rápido sin tener que cargar manualmente los datos. Alternativamente, se puede hacer un enfoque manual: la app recupera los datos de Postgres (por ejemplo vía SQLAlchemy en un DataFrame) y luego usa DuckDB (con su binding Python) para ejecutar consultas SQL sobre ese DataFrame o sobre un archivo Parquet/CSV de ser necesario. DuckDB se integra perfectamente con Python y puede trabajar con DataFrames de Pandas directamente [3], lo cual brinda flexibilidad.

Uso en la Plataforma: En la práctica, para una *consulta de serie histórica* (por ejemplo, el usuario pide graficar Bitcoin último año con media móvil de 30 días), la secuencia sería: 1. La app Flask toma de Postgres los datos de precio de Bitcoin. 2. Crea una conexión DuckDB en memoria y registra el DataFrame (o conecta a Postgres directamente). 3. Ejecuta una consulta SQL en DuckDB para computar los datos requeridos (por ejemplo, añadir columna de media móvil calculada con `AVG` window function sobre 30 días). 4. Retorna el resultado (posiblemente los mismos datos de precios con columnas adicionales) para enviarlos al frontend. DuckDB maneja esta carga de forma muy eficiente, incluso con conjuntos de datos grandes, gracias a su motor optimizado.

Este enfoque híbrido garantiza que el sistema principal de registro (PostgreSQL) permanece ligero y normalizado, mientras que la capa DuckDB proporciona **análisis acelerado** sobre los datos almacenados cuando sea necesario, cumpliendo la meta de agilidad en consultas complejas desde el frontend.

Obtención de Cotizaciones Diarias (API de Criptomonedas)

Para mantener actualizados los precios, la aplicación incorporará integraciones con una API pública de criptomonedas. Se recomienda usar la API de **CoinGecko**, ya que es abierta, no requiere autenticación para uso básico y proporciona datos de miles de monedas. CoinGecko ofrece un plan gratuito que permite hasta 10.000 consultas al mes (límite de 30 por minuto) sin costo [7], más que suficiente dado que solamente se hará **una consulta por día por criptomoneda**. De hecho, CoinGecko es ampliamente utilizado para obtener precios actuales de criptos en proyectos similares [8].

Implementación: - Al **agregar una nueva cripto** (por ejemplo, Bitcoin), la aplicación almacenará su identificador (CoinGecko ID o símbolo). Podría inmediatamente realizar una llamada a la API para obtener datos históricos iniciales (por ejemplo, endpoint `/coins/{id}/market_chart` para obtener precios pasados). - Cada día (mediante una tarea programada o al presionar un botón "Actualizar cotizaciones"), la aplicación hará una petición HTTP (usando `requests` en Python) al endpoint apropiado de CoinGecko para cada criptomoneda seguida. Un endpoint útil es el de **precio actual** (por ejemplo `/simple/price?ids=bitcoin&vs_currencies=usd` devuelve el precio actual). Alternativamente, para registrar un precio diario consistente (ej. precio de cierre diario), se puede llamar a un endpoint de precio histórico en fecha actual. - La respuesta JSON de la API se parsea y se inserta en la tabla de Precios de PostgreSQL. Para evitar duplicados, la aplicación debe asegurarse de que para una combinación cripto-fecha no inserte dos registros (puede usarse una restricción única en DB o lógica de la app que verifique si ya existe). - Si la API de CoinGecko o la red fallan, se debe manejar el error adecuadamente (loggear el evento, reintentar o notificar en la interfaz). Dado que la frecuencia es baja (1 vez por día), una estrategia simple es reintentar la llamada tras unos segundos en caso de fallo.

Proveedor de Datos Alternativo: CoinGecko es preferido por su apertura; sin embargo, la arquitectura permite cambiar de proveedor en el futuro (por ejemplo, CoinMarketCap, Binance API, etc.) simplemente encapsulando la lógica de obtención de precios en un módulo. Si se requiere un proveedor con clave API, se almacenará dicha clave en variable de entorno para mantener la seguridad.

Funcionalidades del Frontend Web (Flask + Bootstrap)

La aplicación web ofrecerá una interfaz accesible desde el navegador, desarrollada con Flask en el backend generando páginas HTML (utilizando Jinja2 como motor de plantillas). El diseño usará **Bootstrap 5** (framework CSS) para lograr un estilo moderno y responsivo sin un gran esfuerzo de CSS personalizado. Se buscará una apariencia tipo *dashboard*, intuitiva para el usuario.

Páginas/Secciones principales: 1. **Página de Inicio / Dashboard**: mostrará un resumen, por ejemplo, lista de criptomonedas seguidas y sus precios más recientes, con opción rápida de actualizar. Podría incluir también un gráfico general o algunos indicadores clave. 2. **Agregar Criptomoneda**: formulario sencillo donde el usuario ingresa el identificador o selecciona de una lista desplegable la criptomoneda que desea comenzar a seguir. Al enviar, Flask procesa la petición: valida la entrada (ej. que la cripto no esté ya en seguimiento, o que el ID existe en la API), guarda en la base de datos, y posiblemente inicia una carga inicial

de datos históricos. Luego redirige de vuelta al dashboard o a la página de detalle. 3. **Actualizar Cotizaciones**: botón o acción (en el dashboard) que desencadena la consulta de precio diario para todas las criptos seguidas (o podría haber un botón por cada cripto para actualizar individualmente). Esta acción invocará la lógica antes descrita de llamar a la API de CoinGecko y almacenar el nuevo precio. La interfaz debería mostrar un mensaje de éxito o error según resultado. 4. **Visualizar Serie Histórica**: al elegir una criptomoneda (por ejemplo, haciendo clic en su nombre en la lista), se carga una página de detalle con el gráfico de la serie histórica de precios almacenados para esa cripto. Aquí es donde entra DuckDB para consultas avanzadas: el backend recuperará los datos históricos y calculará cualquier indicador solicitado antes de renderizar. La página mostrará opciones de rango de fechas, indicadores técnicos a superponer, etc., y la gráfica correspondiente.

Diseño de Interfaz: - Se utilizará **Bootstrap** para una presentación uniforme: cabecera con el título del proyecto y menú, un sidebar o navbar con opciones (Dashboard, Agregar cripto, etc.), y componentes como tablas, formularios y cards para distribuir la información. - El estilo debe ser *responsive* para que pueda verse en distintos dispositivos. Bootstrap facilita esto con su grid system y componentes ya preparados. - Para mantener un diseño limpio y moderno, se pueden emplear iconos (p. ej. de Font Awesome) para botones (un “+” para agregar cripto, un icono de refresh ⟳ para actualizar precios, etc.). - Dado que es un prototipo, se puede usar un tema estándar de Bootstrap; no obstante, dejando abierta la posibilidad de personalización futura con CSS propio o temas Bootstrap.

Gráficas Interactivas de Precios

Una parte crucial del frontend es mostrar las series históricas de precios de forma interactiva y visualmente atractiva. Para ello se integrará una **librería de gráficos en el navegador**, alimentada por los datos proporcionados por Flask (en formato JSON, por ejemplo, vía la plantilla o una petición AJAX). Hay varias opciones viables: - **Chart.js**: Librería JavaScript sencilla para gráficos que se integra fácilmente en HTML5 canvas. Permite gráficos de líneas, velas, etc., y tiene plugins para anotaciones o líneas de tendencia. Podría usarse para mostrar la curva de precios y agregar indicadores (calculados previamente en backend con DuckDB y enviados como series adicionales al gráfico). - **Plotly**: Biblioteca que genera gráficos interactivos con capacidades de zoom, hover, y leyendas dinámicas. Tiene integración con Python (Flask podría renderizar un gráfico Plotly embebido) o se puede usar la versión JS en frontend. Facilita incluir varios conjuntos de datos (ej. precio y media móvil) y es muy útil para **indicadores técnicos**. - **D3.js**: Biblioteca de más bajo nivel pero muy poderosa para visualizaciones personalizadas. D3 permitiría control total para implementar gráficos financieros con detalles como líneas de tendencia dibujadas por el usuario, pero su curva de aprendizaje es más alta. En un inicio, quizá no sea necesario algo tan complejo.

Dado que buscamos interactividad (poder activar/desactivar tendencias, ver valores al pasar el cursor, etc.), **Plotly** o **Chart.js** resultan adecuadas. Flask puede enviar los datos preparados al renderizar la plantilla (por ejemplo, incrustando un JSON con fechas y precios), y un script JS en la página genera el gráfico. Estas bibliotecas son open-source y bastante flexibles; usar Flask + una librería JS de gráficos es una combinación común para dashboards interactivos [2].

Características a implementar en las gráficas: - **Selección de Indicadores**: permitir al usuario agregar elementos como una media móvil de 7 días, 30 días, bandas de Bollinger, etc. Al seleccionar, el frontend podría solicitar (vía AJAX) recalcular usando DuckDB y devolver los puntos necesarios, o ya haber sido calculados en la carga inicial. - **Interactividad**: habilidad de hacer *zoom* en rangos de fechas, ver tooltip con valores exactos al posicionar el cursor, y leyendas claras. Plotly, por ejemplo, ofrece zoom y pan out-of-the-

box. - **Tendencias y Anotaciones**: en esta versión de desarrollo, quizás solo se muestren líneas calculadas (ejemplo: tendencia lineal o máximos/mínimos marcados). En el futuro se podría permitir que el usuario dibuje líneas o marque puntos en el gráfico, pero eso requeriría una capa de interacción más compleja con D3 o una librería especializada. Por ahora, se pueden pre-calcular tendencias (ej. regresión lineal) en backend si se desea y mostrarlas. - **Actualización**: Si se agregan nuevos datos (nueva cotización del día), la gráfica debería reflejarlo. Esto se puede lograr recargando la página de la gráfica o, mejor aún, haciendo que la parte de frontend consulte de nuevo los datos (AJAX) y re-renderice el gráfico sin refrescar toda la página.

Consideraciones de Seguridad

Desde el inicio, la arquitectura debe contemplar la seguridad, especialmente si en el futuro se añade autenticación de usuarios u otras funcionalidades sensibles. Algunas **mejores prácticas** a seguir:

- **Validación de Entradas**: Cualquier dato que ingrese el usuario (p.ej. el identificador de la criptomoneda a agregar) debe ser validado y sanitizado. Esto previene ataques de inyección o XSS. Por ejemplo, si se envía un símbolo de cripto, comprobar que es alfanumérico o coincide con una lista permitida. "Asegúrate de que la entrada de datos de los usuarios está desinfectada y validada para evitar ataques de inyección" [9] . Flask/Jinja escapa por defecto las variables en plantillas para mitigar XSS, pero igualmente revisar cualquier inserción de HTML dinámico.
- **Uso de Consultas Parametrizadas/ORM**: Como se mencionó, nunca construir consultas SQL concatenando strings con datos externos. Siempre usar parámetros (prepare statements) o un ORM como SQLAlchemy que lo gestione automáticamente. Esto protege contra *SQL Injection* [4] . Asimismo, evitar ejecutar comandos del sistema u otras acciones con entradas no validadas.
- **Autenticación y Autorización**: Si bien en esta versión no se implementa un sistema de usuarios final, conviene diseñar la estructura pensando en ello. Por ejemplo, separar las rutas que en el futuro podrían requerir login (agregar cripto, ver datos) de las públicas. Se puede incorporar fácilmente extensiones como **Flask-Login** o **Flask-Security-Too** más adelante. Tener un modelo de usuario en la base de datos desde el principio (aunque no se use de momento) podría ser útil. En todo caso, se recomienda "utilizar métodos seguros de autenticación y garantizar mecanismos adecuados de control de acceso" [10] cuando se implemente esta característica.
- **Comunicación Segura**: Para despliegues en producción, asegurar que el tráfico web sea por **HTTPS** (TLS). Esto implica configurar un certificado SSL (let's encrypt, por ejemplo) en el reverse proxy (Nginx). Adicionalmente, si se expone una API REST para las cotizaciones, considerar protegerla con autenticación (JWT o token) en caso de uso multi-usuario.
- **Gestión de Secretos**: Nunca exponer claves API (si se usaran) o credenciales en el repositorio. Utilizar variables de entorno y posiblemente un gestor de secretos si la infraestructura lo soporta. En Docker Compose, definir las passwords en un `.env` fuera del código.
- **Cabezeras de Seguridad**: Configurar cabeceras HTTP en las respuestas para mitigar ataques comunes. Por ejemplo, usar **Content Security Policy (CSP)** para prevenir inyecciones de scripts, `X-Frame-Options: DENY` para evitar clickjacking, etc. Flask permite configurar esto vía extensiones (Flask-Talisman) o manualmente.
- **Protección CSRF**: Si la aplicación incluye formularios que modifican datos (ej. el formulario de agregar cripto), activar protección anti-CSRF. Flask-WTF extension proporciona tokens CSRF fácilmente. Esto evitará que peticiones maliciosas desde otros sitios puedan engañar al usuario autenticado a realizar acciones no deseadas [11] .

- **Docker Security**: A nivel de contenedores, usar imágenes oficiales y actualizadas (ej. `postgres:15-alpine` o similar, `python:3.11-slim` para la app). Ejecutar los procesos con usuarios no privilegiados dentro del contenedor (no root) para limitar el impacto de compromisos. Publicar solo los puertos necesarios. También, restringir recursos si aplica (memoria/CPU) para evitar DoS simples.
- **Dependencias Seguras**: Mantener actualizadas las bibliotecas Python (por ejemplo, parches de seguridad de Flask o SQLAlchemy). Utilizar versiones específicas (pinning en `requirements.txt`) y escanear vulnerabilidades conocidas (herramientas SCA) regularmente. *"Utiliza bibliotecas bien mantenidas que se actualicen activamente para parchear vulnerabilidades"* [9] .

En resumen, adoptar una mentalidad de *seguridad desde el diseño*. Aunque la aplicación inicial sea simple, estas prácticas sientan las bases para que al crecer o volverse pública, esté protegida contra las amenazas más comunes.

Extensibilidad y Mantenibilidad

Desde su concepción, el proyecto estará preparado para **futuras extensiones** y para facilitar el trabajo de otros desarrolladores que lo continúen. Algunas decisiones para lograr esto:

- **Estructura Modular**: Organizar el código Flask en Blueprints o módulos por funcionalidad. Por ejemplo, un blueprint para la parte de cotizaciones (endpoints de agregar/actualizar), otro para la parte de visualización (endpoints de gráficos), e incluso uno futuro para autenticación. Esto hace más fácil agregar nuevas secciones sin tocar las existentes. Dentro del código, seguir patrones MVC donde posible: modelos (clases o funciones de acceso a DB), lógica (servicios) separada de las rutas de Flask.
- **Documentación Interna**: Incluir documentación para el programador en el código mismo: comentarios explicando piezas complejas, docstrings en funciones y clases, y quizá un `README.md` en el repositorio describiendo cómo desplegar el proyecto, cómo agregar una nueva dependencia, cómo correr las actualizaciones diarias manualmente, etc. Dado que es una "versión de desarrollo", se espera que esté **bien documentada para el programador** que pueda retomarla. Esto incluye también detallar en el README la estructura de archivos, por ejemplo: `app.py` o `wsgi.py` como entrada, carpeta `templates/` para HTML, `static/` para CSS/JS, etc., y cómo se configura Docker.
- **Preparación para Autenticación**: Como ya se mencionó, dejar listo el andamiaje para agregar login de usuarios. Por ejemplo, se podría incluir desde ya un modelo de usuario en la base de datos y quizá un simple formulario de login (aunque sea inoperante de momento) o al menos planificar rutas `/login`, `/logout`, protegiendo ciertas vistas con decoradores que en un futuro chequearán sesión. Esto evitará reestructurar mucho cuando se incorpore autenticación.
- **Facilidad para Nuevas Funcionalidades**: El diseño con DuckDB ya habilita a futuro análisis más sofisticados (incluso podría integrarse con librerías de IA para predicción de precios). Asimismo, el sistema podría extenderse para manejar **portafolios de usuarios** (cada usuario sigue ciertas criptos y ve su propia dashboard), o agregar funcionalidades como alertas (ej. notificar si una cripto sube o baja cierto porcentaje).
- **Bootstrap y Frontend**: Al usar Bootstrap y estándares web, un desarrollador de frontend podría reemplazar o mejorar la interfaz sin tener que modificar el backend. Mantener separada la lógica de presentación (plantillas HTML, archivos JS) de la lógica de negocio facilita trabajar en ello independientemente.

- **API REST**: Considerar que en el futuro se quiera exponer una API JSON para estas funciones (por ejemplo, una ruta REST para "GET /api/prices/bitcoin" que devuelva los datos en JSON). Si se estructura bien el código (usando Flask blueprints, por ejemplo), se puede implementar simultáneamente la funcionalidad HTML y JSON fácilmente. Esto daría versatilidad (un cliente móvil podría usar la API, por decir algo).
- **Pruebas**: Aunque sea un desarrollo inicial, es bueno sentar bases para tests (unitarios o de integración). Por ejemplo, incluir algunos tests para la función que llama a la API externa (simulando una respuesta), o para la lógica de cálculo con DuckDB. Un proyecto con tests es más fácil de extender sin romper funcionalidades existentes.

En definitiva, se busca escribir un código **claro, modular y documentado**, siguiendo principios SOLID en lo posible, de modo que agregar una nueva funcionalidad en el futuro (sea otra estadística de cripto, soporte multi-moneda base, o migración a un entorno escalable) requiera el menor esfuerzo y riesgo posible.

Conclusión

Se han definido las instrucciones y lineamientos para construir un proyecto web de seguimiento de cotizaciones de criptomonedas utilizando Docker, PostgreSQL, Flask (Python) y DuckDB, cumpliendo con las características requeridas. La elección de tecnologías garantiza un desarrollo ágil y un rendimiento robusto: Python/Flask provee simplicidad y potencia en el backend [1], PostgreSQL asegura almacenamiento confiable de los datos diarios, DuckDB ofrece velocidad en el análisis de datos financieros [5], y un frontend con Bootstrap + librerías de gráficas brinda una experiencia interactiva moderna.

Además, la solución se plantea con una atención especial a la seguridad (validación de entradas, buenas prácticas de programación segura, preparación para autenticación) y a la extensibilidad futura. Con esta base, el equipo de desarrollo podrá implementar la plataforma de forma ordenada y documentada, dejando sentadas las bases para expandirla con nuevas funcionalidades en el futuro conforme sea necesario. Los componentes y pasos descritos aquí servirán como guía (*project.md*) para que el Codex CLI (u otra herramienta generativa) pueda crear y estructurar el proyecto adecuadamente, manteniendo un estándar de calidad alto en la arquitectura propuesta.

Fuentes Consultadas:
- Documentación de seguridad en Flask y buenas prácticas de desarrollo en Python [12] [4]
- Tutoriales y artículos sobre integración de DuckDB con Python para análisis de datos financieros [5] [3]
- Uso de la API pública de CoinGecko para obtener precios de criptomonedas (plan gratuito y capacidades) [7] [8]
- Recomendaciones de arquitectura web con Flask, D3.js y librerías de gráficos interactivas [2]

---

[1] [4] [9] [10] [12] Construir Aplicaciones Python Seguras: Buenas prácticas y soluciones de ciberseguridad para empresas modernas - Hodeitek | Ciberseguridad, IA y Servicios Digitales
https://hodeitek.com/es/blog/ciberseguridad/construir-aplicaciones-python-seguras-buenas-practicas-y-soluciones-de-ciberseguridad-para-empresas-modernas/

[2] Build interactive charts with Flask and D3.js - LogRocket Blog
https://blog.logrocket.com/build-interactive-charts-flask-d3js/

3 5 DuckDB Tutorial for Traders – A Python Guide
https://www.marketcalls.in/python/duckdb-tutorial-for-traders-a-python-guide.html

6 Using MotherDuck at MotherDuck: Loading Data from Postgres with DuckDB - MotherDuck Blog
https://motherduck.com/blog/pg%20to%20motherduck%20at%20motherduck/

7 Crypto Data API: Most Comprehensive & Reliable Crypto Price & Market Data | CoinGecko API
https://www.coingecko.com/en/api

8 GitHub - stylepatrick/crypto_price_tracker: Crypto price tracker which stores values in PostgresDB via a scheduler with manual trigger via REST API. Coingecko API is used to get the actual price.
https://github.com/stylepatrick/crypto_price_tracker

11 Consideraciones de Seguridad — Flask Documentation (3.1.x)
https://flask-es.readthedocs.io/web-security/