**Project Plan / Roadmap**

**1. Project Description**

**Goal**: Build a bot that **analyzes newly launched tokens**, checks metrics (market cap, volume, sentiment, “rug” risk, etc.), filters suspicious tokens, and categorizes them by a confidence score (High, Medium, Low).  
**Technologies**:

1. **Language & Framework**: Python (for data fetching, analysis, real-time tracking); optional integration with Solana if on-chain interactions are needed.
2. **APIs & Tools**:
   * LunarCrush for social sentiment
   * RugCheck API for contract safety
   * Market data endpoints for volume / liquidity checks
   * WebSockets (for real-time tracking) or Polling (for periodic checks)

**2. High-Level Requirements**

1. **Real-Time Monitoring**
   * WebSocket or polling-based approach to detect newly launched tokens.
2. **Data Ingestion**
   * APIs to fetch token metadata (price, market cap, transaction volume)
   * Social sentiment from LunarCrush
   * RugCheck safety score
3. **Validation & Filtering**
   * Configurable thresholds for deciding if a token is “safe”
   * Blacklists for known bad tokens or developer addresses
4. **Categorization**
   * Automatic scoring of tokens as High, Medium, or Low confidence
5. **Alerts & Logging**
   * Comprehensive logs for debugging
   * Optionally, real-time alerts via Telegram, email, or Slack
6. **Data Persistence**
   * Caching for quick lookups
   * Optional database for historical analytics (SQLite / Postgres)

**3. Architecture Outline**

1. **Core Components**
   * **Configuration Loader** (config.json or environment variables)
   * **API Handler** (for external requests)
   * **Data Validator** (ensures required fields, correct formats)
   * **Filtering & Scoring Engine** (applies blacklists, thresholds, confidence rules)
   * **Real-Time Tracking Module** (WebSocket or polling)
   * **Caching & Rate Limiter** (avoid repeated API calls and handle rate limits)
2. **Data Flow**
   * **Real-Time Tracker** triggers new token data (via WebSocket or scheduled polling).
   * **Validator** checks required fields.
   * **API Handler** collects additional data (social sentiment, volume, rug check).
   * **Filter & Scoring** applies blacklists, thresholds, and determines final token category.
   * **Caching** stores results for quick reuse.
3. **Infrastructure**
   * **Local / Cloud VM** for running the bot 24/7
   * **Optional** Containerization (Docker) for portability
   * **Database** (if historical or advanced analytics needed)

**4. Phases & Tasks**

**Phase 1: Requirements & Setup**

1. **Gather all API credentials** (e.g., LunarCrush API key, RugCheck URL, etc.)
2. **Define Key Metrics** (market cap, holders, volume thresholds, sentiment thresholds)
3. **Set Up Project Environment**
   * Python 3.x environment
   * Install dependencies (pip install -r requirements.txt)
   * Initialize Git repository for version control

**Deliverables**:

* Finalized config.json with thresholds and API keys
* Basic project skeleton (folders for utils, apis, filters, etc.)

**Phase 2: Data Fetching & Validation**

1. **Implement APIHandler**
   * Create HTTP sessions with retry logic
   * Functions for fetching data from:
     + LunarCrush (social sentiment)
     + RugCheck (safety checks)
     + Market data endpoints (price, volume)
2. **Build utils.py**
   * **validate\_and\_format\_data()**: Ensures all required fields (name, status, confidence, etc.)
   * **handle\_rate\_limit()** decorator for retrying calls
   * **cached\_data()** or LRU cache for repeated lookups
3. **Integrate Logging**
   * Set logging levels in config.json
   * Log successes, warnings, and errors

**Deliverables**:

* Working data fetching from all APIs
* Validation logic that raises errors for missing fields
* Rate-limited, cached calls to avoid excessive requests

**Phase 3: Filtering & Confidence Scoring**

1. **Blacklist & Whitelist Logic**
   * Optionally load blacklisted tokens or dev addresses from file / DB
2. **Confidence Scoring**
   * High, Medium, Low based on thresholds (sentiment score, transaction volume, etc.)
3. **Integration**
   * Combine data from APIs with local checks (e.g., blacklists, thresholds)
   * Provide a single function filter\_data() that processes tokens end-to-end
4. **Optional**: Automatic re-check scheduling or triggers

**Deliverables**:

* A function that takes token data as input, runs all checks, and returns the final category (or discards suspicious tokens)
* Logging for each decision (e.g., “skipped coin XYZ for failing RugCheck”)

**Phase 4: Real-Time Tracking**

1. **WebSocket Approach** (preferred if available)
   * Connect to the platform’s WebSocket
   * Subscribe to new token events
   * Pass each token to the filter pipeline
2. **Polling Approach** (if WebSocket is not an option)
   * Periodically (e.g., every minute) call the platform’s API
   * Compare new token list to previously seen tokens
   * Process newly launched tokens with the filter pipeline
3. **Alerting**
   * If a token passes all checks with a “High” confidence rating, send a Telegram or Slack alert

**Deliverables**:

* A background process or script that continuously listens for new tokens
* Real-time or near real-time detection and processing
* Optional notifications for key events

**Phase 5: Testing**

1. **Unit Tests**
   * For validate\_and\_format\_data, caching, rate-limit decorator, etc.
2. **Integration Tests**
   * Mock external APIs to simulate different scenarios (API error, slow response, partial data)
   * Verify the pipeline discards invalid tokens and labels valid ones correctly
3. **Performance Tests**
   * High volume of tokens (simulate hundreds) to ensure system performance

**Deliverables**:

* Automated test suite (e.g., pytest)
* Consistent passing results for stable code

**Phase 6: Deployment & Scaling**

1. **Production Environment**
   * Configure a cloud server (AWS, GCP, or similar) or host on a local machine with stable internet
2. **Dockerization** (optional)
   * Containerize the application to simplify deployment
   * Potentially orchestrate multiple containers if usage grows
3. **Continuous Integration / Continuous Deployment (CI/CD)**
   * Set up GitHub Actions or GitLab CI to automate testing and deployment
4. **Monitoring**
   * Track logs, error rates, and system health
   * Use a monitoring stack (e.g., Prometheus + Grafana) if required

**Deliverables**:

* A running production instance, accessible 24/7
* Automated system to handle new code merges, run tests, and deploy if successful

**Phase 7: Advanced Features & Maintenance**

1. **Database Integration**
   * Switch from in-memory caching to SQLite or PostgreSQL for long-term storage and historical analytics
2. **Machine Learning / Predictive Analysis**
   * Train models using historical data to estimate future token performance or rug risk
3. **User Interface**
   * Web dashboard to visualize top coins, logs, and analysis results
   * Telegram or Slack bot that responds to commands like /check <token>
4. **Security & Hardening**
   * Secure API keys in environment variables or a secrets manager
   * Validate input thoroughly against potential injection or malicious data
5. **Continuous Maintenance**
   * Update dependencies, handle new API endpoints, incorporate user feedback

**Deliverables**:

* Enhanced, data-driven insights (e.g., ML-based scoring)
* More robust user access (web or bot-based)
* Sustained updates and improvements over time

**Conclusion**

This plan guides the development of a **token analysis bot** that pulls real-time data from multiple APIs, validates and filters tokens, assigns confidence categories, and optionally alerts on high-potential coins. It covers everything from initial setup and configuration to advanced enhancements like machine learning and a user-facing interface.

Feel free to tailor any step to your exact business logic or technology stack, then feed this roadmap into your custom AI to generate and orchestrate tasks systematically.

**You said:**

add using solscans rpc and websockets

Reasoned for a few seconds

Below is an **updated Project Plan / Roadmap** that specifically incorporates **SolScan’s RPC and WebSocket** endpoints for real-time tracking on Solana. This plan is designed so you can feed it into a custom AI that specializes in planning and building projects from given specifications. Feel free to copy/paste as a single prompt, and modify details as needed.

**Project Plan / Roadmap (SolScan Integration Version)**

**1. Project Description**

**Goal**:  
Build a bot to **analyze newly launched or updated Solana tokens** in real-time by connecting to **SolScan’s RPC/WebSocket** endpoints. The bot checks market cap, volume, sentiment (via other APIs), and detects “rug” or scam risks, ultimately categorizing tokens with a confidence score (High, Medium, Low).

**Key Technologies**:

1. **SolScan’s RPC / WebSocket** for real-time Solana data
2. **Python** (primary language) for data orchestration
3. **Anchor / Rust** (if on-chain Solana program integration is required)
4. **Additional APIs** (e.g., LunarCrush, RugCheck) for cross-referencing token legitimacy

**2. High-Level Requirements**

1. **Real-Time Solana Tracking**
   * Use **SolScan’s WebSocket** for event notifications (e.g., new token mint)
   * Optionally leverage direct **Solana JSON RPC** subscriptions (logs, signatures, or token-related instructions)
2. **Token Data Collection**
   * Market metrics from various APIs
   * Social sentiment from LunarCrush
   * Rug safety checks (RugCheck or other services)
3. **Validation & Filtering**
   * Blacklist known scam tokens or dev addresses
   * Threshold-based filters (volume, holders, etc.)
4. **Confidence Scoring**
   * Categorize tokens as High, Medium, or Low confidence using dynamic thresholds
5. **Alerts & Logging**
   * Logs for debugging and optional immediate alerting (Telegram, Slack, etc.)
6. **Persistence & Caching**
   * Caching in memory for quick lookups
   * Potential database for historical stats

**3. Architecture Outline**

1. **Core Components**
   * **Configuration Loader** (config.json)
   * **SolScan RPC/WebSocket Handler** (handles event-based or RPC data from SolScan)
   * **Additional API Integrations** (LunarCrush, RugCheck, custom volume endpoints)
   * **Filtering & Scoring** (applying thresholds, blacklists, and confidence levels)
   * **Caching & Rate Limiting** (prevent excessive API calls)
2. **Data Flow**
   * **SolScan WebSocket** notifies the system about new token events (e.g., minted tokens, token metadata changes).
   * The system **fetches** additional data from relevant APIs (market metrics, social sentiment).
   * The **filter and scoring** engine applies blacklists, thresholds, calculates final confidence.
   * **Caching** ensures repeated queries don’t overload external APIs.
3. **Infrastructure**
   * **Local / Cloud VM** or containerized environment
   * **Database** (SQLite or PostgreSQL) if persistent historical data is needed

**4. Phases & Tasks**

**Phase 1: Requirements & Setup**

1. **Collect All Access Details**
   * SolScan RPC/WebSocket credentials or endpoints
   * Other API keys (LunarCrush, RugCheck, etc.)
2. **Define Analysis Metrics**
   * Minimum market cap, holder count, sentiment threshold
   * Specific fields from SolScan (e.g., program logs, mint addresses)
3. **Project Environment**
   * Python 3.x environment
   * Set up requirements.txt for all libraries (e.g., websockets, requests)
   * Git repo for version control

**Deliverables**:

* Initial config.json with details for SolScan endpoints, thresholds, and logging preferences
* Basic folder structure (apis/, filters/, solana\_tracker/, etc.)

**Phase 2: SolScan Integration**

1. **WebSocket Handler**
   * **Connect to SolScan’s WebSocket** for real-time blockchain events
   * Subscribe to relevant channels (e.g., token mint notifications, token metadata updates)
2. **RPC Fallback**
   * Implement periodic queries using SolScan’s RPC as a fallback or for confirmations
3. **Event Processing**
   * Parse incoming data to extract token addresses or relevant details
   * Queue or hand off events to the data validation pipeline

**Deliverables**:

* **solana\_tracker.py** (or similar) that encapsulates the WebSocket connection logic
* Basic event loops that log new tokens or changes in real time

**Phase 3: Data Fetching & Validation**

1. **APIs & Rate Limiting**
   * Python requests or aiohttp to call RugCheck, LunarCrush, etc.
   * Implement handle\_rate\_limit decorators
2. **Validation**
   * validate\_and\_format\_data() ensures data has mandatory fields (token address, dev, volume, etc.)
   * Check for unexpected or missing values
3. **Caching**
   * Use lru\_cache or a caching library to store results for quick lookups

**Deliverables**:

* End-to-end data flow for newly discovered tokens from SolScan’s WebSocket → external API calls → validated data

**Phase 4: Filtering & Scoring**

1. **Blacklist / Whitelist**
   * Maintain a list of suspicious token addresses or dev addresses
2. **Threshold Checks**
   * Market cap, holder count, transaction volume, dev track record, etc.
3. **Confidence Score**
   * If above a certain sentiment score and volume threshold, label as “High”
   * Otherwise, Medium or Low
4. **Output**
   * Logs indicating the token’s final classification
   * Optional real-time notifications via Telegram, Slack, or email

**Deliverables**:

* **filters.py** that provides filter\_data() or determine\_confidence(token)
* Thorough logs for acceptance or rejection of each token

**Phase 5: Testing & Reliability**

1. **Unit Tests**
   * Check logic for real-time tracking, validation, filtering, caching
2. **Integration Tests**
   * Mock SolScan events (WebSocket messages)
   * Mock external API calls (RugCheck, LunarCrush) to test combined flows
3. **Performance & Stress Tests**
   * Burst of events from SolScan to ensure the system can handle spikes

**Deliverables**:

* Automated test suite (e.g., pytest)
* Documented test results for reliability under typical or high-load conditions

**Phase 6: Deployment & Scaling**

1. **Production Setup**
   * Deploy the bot on a server with stable connectivity to SolScan’s WebSocket
   * Potential Dockerization for consistent environments
2. **CI/CD Integration**
   * GitHub Actions or GitLab CI for automated testing and deployment
3. **Monitoring & Alerts**
   * Watch logs for errors or disconnections
   * Reconnect logic if WebSocket fails

**Deliverables**:

* Live, 24/7 running service that tracks new Solana tokens in real-time
* Automated pipeline that updates, tests, and deploys on code changes

**Phase 7: Advanced Extensions**

1. **Historical Analysis & DB Storage**
   * Store token data in SQLite/Postgres for long-term analytics
   * Generate time-series insights
2. **Machine Learning**
   * Predict future token performance based on historical data
   * Potentially incorporate on-chain transaction patterns
3. **User Interface**
   * Web dashboard to display top new tokens, logs, and metrics
   * Telegram command-based interface (e.g., “/check <token\_address>”)

**Deliverables**:

* Extended analytics or UI/UX improvements
* Ability to maintain and enhance system as Solana evolves

**5. Timeline & Milestones**

1. **Week 1-2**: Setup environment, plan architecture, integrate basic SolScan WebSocket connection
2. **Week 3-4**: Implement external API calls, validation, caching, and initial filters
3. **Week 5-6**: Finalize confidence scoring, real-time alerts, and blacklisting logic; begin testing
4. **Week 7**: Deploy to production environment (Docker or direct VM); set up CI/CD
5. **Week 8+**: Enhance with database storage, user interface, or machine learning as needed

**6. Conclusion**

By leveraging **SolScan’s RPC and WebSocket** for real-time updates on Solana token events, alongside supplemental data from external APIs (RugCheck, LunarCrush, etc.), this bot can effectively **track new or updated tokens, filter out bad actors, and highlight promising opportunities**. Follow the phases above for a structured build-out, ensuring each component is rigorously tested and easily extensible for future features.