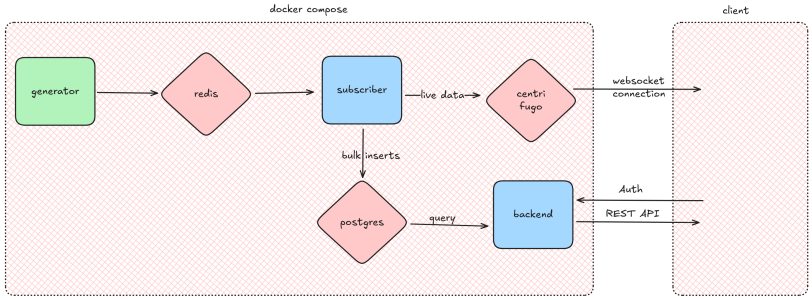
Baryonic Space Backend Engineer Task

This task is designed to assess your ability to build, deploy, and maintain a distributed system. It is divided into two main steps. Step 1 focuses on building the core application logic. Step 2 focuses on making the system more robust, secure, and observable, reflecting real-world production requirements.



The diagram above illustrates the target architecture for Step 1. The final system, after completing Step 2, will build upon this diagram by introducing Nginx, Vector, and Grafana.

Step 1: Core System Implementation

Overview

The objective of this initial step is to develop a system for real-time and retrospective monitoring of time-series data. The system will comprise six distinct components, as illustrated in the diagram.

The system is expected to consist of Dockerized components and be deployable with a single `docker compose up` command. Three of these components—Redis, PostgreSQL, and Centrifugo—can be directly pulled from Docker Hub. The other three—`generator`, `subscriber`, and `backend`—will need their own Dockerfiles. The `generator` service is provided; your work is scoped to the **`subscriber`** and **`backend`** services.

You should submit your solution as a zip file that includes a main `docker-compose.yaml` file to orchestrate the services and a detailed `README.md`.

Component Details (Step 1)

Generator Service (Provided)

- This service creates two Redis channels, `NYSE` and `NASDAQ`, and publishes mock stock prices to each.

- It should run automatically with `docker compose up`.
- **Verification**: You can verify its operation by executing `docker exec -it <redis_container_name> redis-cli` and then `SUBSCRIBE NYSE`. You should see a stream of messages like `"TSLA:723.02"`.

Subscriber Service (To Be Implemented)

- **Task**: Implement this service in Python or Go.
- **Functionality**:
  1. Listen to both the `NYSE` and `NASDAQ` Redis channels.
  2. Forward incoming data to the **Centrifugo** service via its API for real-time updates.
  3. Batch incoming data and perform bulk inserts into the **PostgreSQL** database at a fixed interval (e.g., every 5 seconds) to handle the write load efficiently.
- **Libraries**: You can use libraries like `pycent` (Python) or `centrifuge-go` (Go) for Centrifugo, and `psycopg` (Python) or the standard `database/sql` package (Go) for PostgreSQL.

PostgreSQL Service (Configuration)

- **Task**: Configure the official PostgreSQL Docker image.
- **Initialization**: Use the `/docker-entrypoint-initdb.d` directory to run an initial SQL script that creates the necessary table(s).
- **Schema Example**:

```sql
CREATE TABLE stock_prices (
    timestamp TIMESTAMPTZ NOT NULL,
    stock_name VARCHAR(255) NOT NULL,
    exchange_name VARCHAR(255) NOT NULL,
    price NUMERIC NOT NULL
);
-- Consider creating an index for performance on timestamp and stock_name
CREATE INDEX idx_stock_prices_ts_name ON stock_prices (stock_name, timestamp DESC);
```

Centrifugo Service (Configuration)

- **Task**: Configure the official Centrifugo Docker image.
- **Configuration**: Since no authentication is required, you must set the appropriate flags in its configuration file (e.g., `config.json`) to allow anonymous connections and subscriptions.
  - Search for `allow_anonymous_connect_without_token` and `allow_subscribe_for_anonymous` in the recent Centrifugo documentation. The exact keys might vary depending on the centrifugo version you are using.

Backend Service (To Be Implemented)

- **Task**: Implement a simple REST API using a Python framework like **Django** (with `django-ninja` or `DRF`) or **FastAPI**.
- **Endpoints**:
  1. `GET /api/prices/{stock_name}`: Retrieves all price points for a given stock between a `start_time` and `end_time` query parameter.
  2. `GET /api/average/{stock_name}`: Calculates and returns the average price for a given stock between a `start_time` and `end_time`.
- **Auth**: No authentication or authorization is needed.

Frontend Client (To Be Implemented)

- **Task**: Create a minimal HTML/JavaScript client. UI/UX is not important.
- **Functionality**:
  1. A section to display **live** stock prices by connecting to the Centrifugo WebSocket endpoint.
  2. A form with inputs for a stock name, start time, and end time, and a button that queries the `backend` service to display historical data.

Step 2: Production-Readiness and Observability

After completing Step 1, you have a functional application. This second step challenges you to make it more secure, manageable, and observable.

1. Nginx Reverse Proxy

Your system currently exposes multiple ports. In a production environment, all traffic should flow through a single, secure entry point.

- **Task**: Introduce Nginx as a reverse proxy for the entire system.
- **Requirements**:
  - Add an `nginx` service to your `docker-compose.yaml`.
  - Create a `nginx.conf` file to handle request routing.
  - **Securely expose services**: Only the Nginx port (e.g., `80`) should be exposed to the host machine in your `docker-compose.yaml`. All other services (`backend`, `centrifugo`, `grafana`) should only be accessible within the Docker network.
  - **Routing**:
    - Requests to `/api/*` should be proxied to the `backend` service.
    - WebSocket connections for Centrifugo (e.g., to `/connection/websocket`) should be correctly proxied to the `centrifugo` service.
    - Requests to `/` should serve the `frontend` client.
  - Update your frontend client to make API and WebSocket requests to the Nginx proxy, not the individual services.

2. Observability with Vector

To understand how the system is behaving, you need to collect metrics.

- **Task**: Use Vector.dev to add observability to your system.
- **Requirements**:
  - Add a `vector` service to your `docker-compose.yaml`.
  - Create a `vector.toml` (or `.yaml`) configuration file.
  - **Select and Implement a Metric**: Your primary task is to choose a meaningful metric for this application, instrument your code to generate it, and configure Vector to process it.
    - **Examples of suitable metrics**:
      - The latency (duration) of bulk inserts into PostgreSQL.
      - The error count for each service.
    - You will need to modify one of your services (the `subscriber` is a good candidate) to output this metric data, for example, as a structured log (JSON) to `stdout`.
  - **Configure Vector**:
    - Use a `source` (e.g., `docker_logs`) to collect the metric data.
    - Use a `transform` (e.g., `remap`) if you need to parse or structure the data.
    - Use a `sink` (e.g., `sql`) to **save the collected metric into a new table in your PostgreSQL database**. You will need to create this new table.

Bonus Tasks

Completing these bonus tasks will demonstrate a deeper understanding of the technologies involved.

Bonus 1 (Grafana Dashboard)

Visualize the metric you created in Step 2.

- **Task**: Add Grafana to the stack to create a monitoring dashboard.
- **Requirements**:
  - Add a `grafana` service to your `docker-compose.yaml`.
  - Proxy Grafana through Nginx (e.g., at the `/grafana` location).
  - **Automated Provisioning**:
    - Configure a PostgreSQL data source for Grafana automatically using provisioning files.
    - Provision a dashboard from a JSON file. The dashboard should be mounted into the container and loaded on startup.
    - The dashboard must contain at least one panel that visualizes the metric you collected with Vector and stored in PostgreSQL.

Bonus 2 (PostgreSQL Performance Tuning)

A default PostgreSQL configuration is not optimized for a specific workload.

- **Task**: Tune PostgreSQL's configuration for this application's architecture (which is write-heavy from the `subscriber` and read-heavy for specific queries from the `backend`).
- **Requirements**:
  - Identify key PostgreSQL parameters to tune (e.g., `shared_buffers`, `effective_cache_size`, `maintenance_work_mem`, `wal_buffers`).
  - **Dynamic Configuration**: The tuned settings must not be hard-coded. They should be passed into the PostgreSQL container as **environment variables** from your `docker-compose.yaml` (or an associated `.env` file). This allows the configuration to be easily adapted to different host machines.
  - **Justify your choices**: In your `README.md`, explain which settings you tuned and why you chose the values you did, considering the application's workload.

Final Notes

- Submit all your work, even if you can't manage to complete every part.
- You are encouraged to use AI assistance wherever you need.
- You are advised to start working on the task as early as possible. Proceed step by step.
- **Crucially, include a `README.md` document** with your work that briefly explains your implementation, design decisions for each step, and instructions on how to run the project.
- If there is anything vague, do not hesitate to contact us.