Building a **comprehensive cricket analytics dashboard** with live data from **Cricbuzz**, a **SQL database**, and an **interactive web frontend** involves multiple components. Here's a step-by-step guide to architect and build such a platform:

**🔧 Tech Stack Recommendation**

| **Component** | **Suggested Tools** |
| --- | --- |
| **Frontend** | React.js (for interactive UI) |
| **Backend/API** | Node.js + Express / Python + FastAPI |
| **Database** | PostgreSQL / MySQL |
| **Live Data Integration** | Unofficial Cricbuzz API (scraping or third-party) |
| **Analytics** | SQL queries + visualization libraries (e.g., Chart.js, D3.js) |
| **Authentication** | JWT + OAuth (optional for user-based dashboards) |
| **Hosting** | Vercel/Netlify (frontend), Heroku/Render (backend) |
| **Optional Admin UI** | React Admin / Hasura for easy CRUD |

**🧱 1. System Architecture Overview**

┌────────────┐ ┌──────────────┐ ┌────────────┐

│ Frontend │◄────►│ REST API │◄────►│ SQL DB │

│ (React.js) │ │ (Node/FastAPI│ │ PostgreSQL │

└────┬───────┘ └────┬─────────┘ └────┬───────┘

│ Live UI Updates │ Data Insert/Query ▲

▼ ▼ │

┌──────────────┐ ┌──────────────┐ ┌──────────────┐

│ Cricbuzz API │►──│ Parser Layer │─────────►│ Analytics Job│

└──────────────┘ └──────────────┘ └──────────────┘

**🚀 Step-by-Step Guide**

**🔌 2. Integrate Cricbuzz Live API (Unofficial)**

There is no official Cricbuzz API, but data can be pulled using **web scraping** or **unofficial APIs** like:

* [cricbuzz-api](https://www.npmjs.com/package/cricbuzz)
* or scrape JSON endpoints using requests + BeautifulSoup or axios + cheerio

**Node.js Example** (using cricbuzz npm):

npm install cricbuzz

const cricbuzz = require('cricbuzz');

cricbuzz.getMatches().then(data => {

console.log(data);

});

Store relevant data in the SQL database at intervals (e.g., every 60 seconds) via a background job (cron or scheduler).

**🧮 3. SQL Database Design**

**Tables:**

-- Match Info

CREATE TABLE matches (

id SERIAL PRIMARY KEY,

match\_id VARCHAR UNIQUE,

team1 VARCHAR,

team2 VARCHAR,

status VARCHAR,

start\_time TIMESTAMP

);

-- Player Stats

CREATE TABLE players (

id SERIAL PRIMARY KEY,

player\_id VARCHAR UNIQUE,

name VARCHAR,

team VARCHAR,

role VARCHAR

);

CREATE TABLE player\_stats (

id SERIAL PRIMARY KEY,

match\_id VARCHAR,

player\_id VARCHAR,

runs INT,

balls\_faced INT,

wickets INT,

overs FLOAT,

catches INT,

FOREIGN KEY (player\_id) REFERENCES players(player\_id)

);

-- Live Score

CREATE TABLE live\_scores (

id SERIAL PRIMARY KEY,

match\_id VARCHAR,

team VARCHAR,

runs INT,

wickets INT,

overs FLOAT,

last\_updated TIMESTAMP

);

**⚙️ 4. Backend API (Node.js / FastAPI)**

Create RESTful APIs for:

* **Live match updates**: /api/live
* **Player stats**: /api/players/:id
* **SQL-powered analytics**: /api/analytics/top-scorers
* **CRUD operations**:
  + Create: POST /api/matches
  + Read: GET /api/matches
  + Update: PUT /api/matches/:id
  + Delete: DELETE /api/matches/:id

Example in **Express (Node.js)**:

app.get('/api/players/:id', async (req, res) => {

const { id } = req.params;

const player = await db.query('SELECT \* FROM players WHERE player\_id = $1', [id]);

res.json(player.rows[0]);

});

**💻 5. Frontend (React.js) + Real-Time Updates**

Use axios or fetch to get data from your API.

Install dependencies:

npm install axios react-chartjs-2 chart.js

**Example: Fetch Live Scores**

useEffect(() => {

const fetchData = async () => {

const res = await axios.get('/api/live');

setLiveScore(res.data);

};

fetchData();

const interval = setInterval(fetchData, 60000);

return () => clearInterval(interval);

}, []);

**Charts for Analytics:**

<Bar

data={{

labels: topScorers.map(p => p.name),

datasets: [{

label: 'Runs',

data: topScorers.map(p => p.runs),

}]

}}

/>

**🔍 6. SQL-Driven Analytics**

Run queries like:

* **Top 5 run scorers in current tournament**:

SELECT p.name, SUM(ps.runs) as total\_runs

FROM player\_stats ps

JOIN players p ON ps.player\_id = p.player\_id

GROUP BY p.name

ORDER BY total\_runs DESC

LIMIT 5;

* **Bowling economy leaderboard**:

SELECT p.name, SUM(ps.runs) / SUM(ps.overs) AS economy

FROM player\_stats ps

JOIN players p ON ps.player\_id = p.player\_id

WHERE ps.overs > 5

GROUP BY p.name

ORDER BY economy ASC

LIMIT 10;

Expose these via API endpoints and call from React.

**🛠️ 7. CRUD Functionality**

In the admin dashboard, allow:

* Adding new matches or players
* Updating stats manually (if required)
* Deleting old records
* Editing data via forms

Example using react-hook-form or Formik for form handling.

**🔐 8. Optional: Authentication & Permissions**

Use JWT for token-based auth:

* Only allow admins to access CRUD routes
* Viewers can see analytics and live scores

**📦 9. Deployment**

* **Frontend:** Vercel or Netlify
* **Backend:** Render / Heroku / Fly.io
* **Database:** Supabase / Railway / PlanetScale

Use **cron jobs** (e.g., GitHub Actions, Node node-cron, or external services) to fetch live data regularly and update the database.

**✅ Final Features Checklist**

| **Feature** | **Status** |
| --- | --- |
| ⚡ Real-time match updates | ✅ via API polling |
| 📊 Player statistics | ✅ with historical SQL records |
| 🔍 SQL-driven analytics | ✅ via custom SQL queries |
| 🛠️ Full CRUD operations | ✅ via RESTful backend |

Would you like:

* A **starter GitHub repo** with code?
* A **Dockerized environment**?
* Example **ERD (Entity Relationship Diagram)**?
* Help finding a stable Cricbuzz source?

Let me know how deep you'd like to go.

1. **API Integration**
   * Utilize [Cricbuzz Cricket API](https://rapidapi.com/cricketapilive/api/cricbuzz-cricket/) via REST

Using the **Cricbuzz Cricket API via REST** can be tricky because **Cricbuzz does not provide an official public REST API**. However, developers often rely on **unofficial APIs** or **scraping Cricbuzz's JSON endpoints** to get live cricket data. Below is a practical guide to help you **safely and efficiently utilize Cricbuzz data via REST**:

**⚠️ DISCLAIMER**

* Cricbuzz's data is protected; scraping or using unofficial APIs must comply with their [Terms of Service](https://www.cricbuzz.com/info/terms).
* This guide is **for educational and personal project purposes only**.

**✅ Option 1: Use Unofficial Node.js API (Easy)**

One of the easiest ways to access Cricbuzz data is via the NPM package: [cricbuzz](https://www.npmjs.com/package/cricbuzz)

**🔧 Installation**

npm install cricbuzz

**🔌 Usage Example (Node.js)**

const cricbuzz = require('cricbuzz');

// Get Live Matches

cricbuzz.getMatches()

.then(data => {

console.log("Live Matches: ", data);

})

.catch(err => console.error(err));

// Get Scorecard

cricbuzz.getScorecard('match\_id')

.then(scorecard => {

console.log(scorecard);

});

🔍 You need to extract a valid match\_id from the getMatches() result.

**✅ Option 2: Scrape Cricbuzz JSON Endpoints (Advanced)**

Cricbuzz often serves **live JSON data** to their own frontend via hidden endpoints.

**🔍 How to Find API-like Endpoints:**

1. Open [**https://www.cricbuzz.com/**](https://www.cricbuzz.com/).
2. Use **Dev Tools (Network Tab)** and filter by **XHR**.
3. Look for calls like:

https://www.cricbuzz.com/match-api/livematches.json

https://www.cricbuzz.com/match-api/<match-id>/commentary.json

https://www.cricbuzz.com/match-api/<match-id>/scorecard.json

These endpoints return **JSON data**, which can be parsed easily.

**🧪 Example: Fetch Live Matches in Python**

import requests

url = "https://www.cricbuzz.com/match-api/livematches.json"

headers = {"User-Agent": "Mozilla/5.0"}

response = requests.get(url, headers=headers)

data = response.json()

# Display all live match IDs and titles

for match\_id, match in data.items():

print(f"{match\_id}: {match['header']['match\_desc']} between {match['team1']['name']} and {match['team2']['name']}")

**🔁 Refresh Interval**

* **Polling Interval**: 30–60 seconds is reasonable for live updates.
* Use a **cron job** or **backend service** to poll and store data in your database.

**✅ Sample Cricbuzz REST Endpoint List**

| **Data Type** | **URL** |
| --- | --- |
| All live matches | https://www.cricbuzz.com/match-api/livematches.json |
| Scorecard | https://www.cricbuzz.com/match-api/<match\_id>/scorecard.json |
| Commentary | https://www.cricbuzz.com/match-api/<match\_id>/commentary.json |
| Player profile | https://www.cricbuzz.com/api/player/stats/<player\_id> (rare) |

**📡 Wrap API in Your Own Backend (Optional)**

To expose Cricbuzz data to your frontend safely:

**Example Express.js Endpoint:**

app.get('/api/live-matches', async (req, res) => {

const response = await fetch("https://www.cricbuzz.com/match-api/livematches.json");

const data = await response.json();

res.json(data);

});

🚫 Don’t expose Cricbuzz URLs directly to the frontend — use a proxy backend to avoid CORS issues and legal exposure.

**🛠️ Best Practices**

| **Task** | **Recommendation** |
| --- | --- |
| Caching | Store match data in SQL/Redis to avoid repeated hits |
| Throttling | Limit how often you fetch data |
| Data Modeling | Normalize JSON into matches, teams, players, events tables |
| Error Handling | Gracefully handle timeouts, invalid match IDs |
| Legality | Use only for personal, non-commercial projects unless licensed |

**✅ Summary**

| **Method** | **Pros** | **Cons** |
| --- | --- | --- |
| **Unofficial NPM cricbuzz** | Easy to use | Limited features |
| **Scrape JSON endpoints** | Real-time, rich data | Fragile, may break |
| **Build own REST wrapper** | Secure, scalable | More work |

**🧰 Want Help Setting Up a Full Wrapper?**

I can help you:

* Write a full Express/Python API
* Create a cron job to poll data
* Build a live scoreboard with React
* Normalize Cricbuzz JSON into a SQL database

Just say the word.

* **Implement Python's requests library for data fetching**
* **Handle real-time match data, player stats, and series information**

Great! Let's build a **Python module** using the requests library to:

1. 🔄 Fetch **real-time match data**
2. 📊 Retrieve **player statistics**
3. 🏆 Get **series information**

We'll simulate interaction with **Cricbuzz's unofficial JSON API endpoints**.

**✅ 1. Setup**

Install requests if you haven't already:

pip install requests

**📁 File Structure (suggested)**

cricbuzz\_api/

├── \_\_init\_\_.py

├── cricbuzz.py ← Main logic

└── utils.py ← Optional helper functions

**🐍 cricbuzz.py – Main API Wrapper**

import requests

import time

BASE\_HEADERS = {

"User-Agent": "Mozilla/5.0"

}

class CricbuzzAPI:

def \_\_init\_\_(self):

self.base\_url = "https://www.cricbuzz.com/match-api"

def get\_live\_matches(self):

"""Fetch all live match data."""

url = f"{self.base\_url}/livematches.json"

try:

response = requests.get(url, headers=BASE\_HEADERS)

response.raise\_for\_status()

return response.json()

except requests.RequestException as e:

print(f"Error fetching live matches: {e}")

return {}

def get\_match\_scorecard(self, match\_id):

"""Fetch scorecard for a specific match."""

url = f"{self.base\_url}/{match\_id}/scorecard.json"

try:

response = requests.get(url, headers=BASE\_HEADERS)

response.raise\_for\_status()

return response.json()

except requests.RequestException as e:

print(f"Error fetching scorecard: {e}")

return {}

def get\_match\_commentary(self, match\_id):

"""Fetch live commentary for a match."""

url = f"{self.base\_url}/{match\_id}/commentary.json"

try:

response = requests.get(url, headers=BASE\_HEADERS)

response.raise\_for\_status()

return response.json()

except requests.RequestException as e:

print(f"Error fetching commentary: {e}")

return {}

def get\_series\_info(self, series\_id):

"""Fetch series-level data (example: points table or schedule)."""

url = f"https://www.cricbuzz.com/api/series/{series\_id}" # This endpoint is not public and may not work

try:

response = requests.get(url, headers=BASE\_HEADERS)

response.raise\_for\_status()

return response.json()

except requests.RequestException as e:

print(f"Error fetching series info: {e}")

return {}

def get\_player\_stats(self, player\_id):

"""Fetch player stats (experimental – may not work consistently)."""

url = f"https://www.cricbuzz.com/api/player/stats/{player\_id}"

try:

response = requests.get(url, headers=BASE\_HEADERS)

response.raise\_for\_status()

return response.json()

except requests.RequestException as e:

print(f"Error fetching player stats: {e}")

return {}

**🧪 Example Usage**

from cricbuzz import CricbuzzAPI

cricbuzz = CricbuzzAPI()

# 1. Get Live Matches

live\_matches = cricbuzz.get\_live\_matches()

for match\_id, match\_data in live\_matches.items():

print(f"{match\_id}: {match\_data['header']['match\_desc']} - {match\_data['team1']['name']} vs {match\_data['team2']['name']}")

# 2. Pick a match\_id and get scorecard

some\_match\_id = list(live\_matches.keys())[0]

scorecard = cricbuzz.get\_match\_scorecard(some\_match\_id)

print(scorecard['scorecard'][0]['batting']) # Batting stats

# 3. Get Commentary

commentary = cricbuzz.get\_match\_commentary(some\_match\_id)

print(commentary['comm\_lines'][0]['comm'])

# 4. (Optional) Get Player Stats

player\_stats = cricbuzz.get\_player\_stats("12345") # Replace with actual player\_id

print(player\_stats)

**🧠 Notes**

* **Match ID**: Comes from the livematches.json response. Example: 73902
* **Player ID**: Hard to extract without scraping profile URLs. You may need to manually identify it.
* **Series Info**: Unofficial and endpoint may not always return clean data.
* All requests should use a **user-agent header** to mimic a browser and avoid blocks.
* Use **caching** or **SQLite/Postgres** to reduce repeated API hits.

**🛠️ Optional: Real-Time Polling Script**

def start\_live\_polling(interval=60):

cricbuzz = CricbuzzAPI()

while True:

data = cricbuzz.get\_live\_matches()

print(f"Fetched {len(data)} live matches at {time.ctime()}")

time.sleep(interval)

# Call this to start polling

# start\_live\_polling()

**✅ Next Step?**

Would you like me to:

* 📦 Turn this into a **Python package / Flask API**?
* 🛠️ Build a **SQL schema + data inserter**?
* 📈 Hook it to a **React.js frontend**?

Let me know what direction you'd like to go in next.

**Part 2**

**🏏 Cricbuzz Analytics Dashboard — Streamlit + SQL (End‑to‑End Guide)**

This guide walks you through building a **production‑grade, multi‑page Streamlit web app** that integrates a **Cricbuzz‑style REST API** with a **SQL database**. You’ll get:

* **Real‑time match updates**
* **Detailed player statistics**
* **SQL‑driven analytics and a built‑in SQL Lab**
* **Full CRUD** (Create/Read/Update/Delete) for administrative data management
* **Database‑agnostic design** (PostgreSQL, MySQL, SQLite)

**Important**: Ensure you have a legitimate, authorized data source and you comply with its Terms of Service. This guide assumes you have access to **REST endpoints** that expose match, player, team, and series data in JSON (official, partner, or a proxy service). If using a third‑party wrapper, validate licensing/usage limits.

**1) System Architecture (What we’re building)**

**Client (Streamlit multi‑page app)**

* Live Match page → polls API → stores/reads DB → auto‑refresh scoreboard
* Player Explorer → query DB & chart stats
* SQL Lab → run read‑only SQL with result/download
* Admin → CRUD forms (players/teams/matches)

**Services (Python)**

* CricbuzzClient (requests) → rate‑limited, retrying REST client
* IngestionService → normalize JSON, upsert into DB, incremental refresh

**Data Layer (SQLAlchemy)**

* DB‑agnostic connection factory
* Canonical relational schema + indexes
* Safe parameterized queries

**Caching & Perf**

* st.cache\_resource for DB engine
* st.cache\_data with TTL for API and heavy queries
* Incremental sync keyed by updated\_at/last\_seen

**2) Tech Stack & Prereqs**

* **Python** 3.10+
* **Streamlit** for UI
* **SQLAlchemy** for DB‑agnostic SQL
* **Requests** for HTTP
* **Pydantic** for config
* **Pandas** for tabular display/exports
* (Optional) **Alembic** for migrations, **Docker** for deployment

# ✅ Create & activate a virtual environment

python -m venv .venv && source .venv/bin/activate # Windows: .venv\Scripts\activate

# ✅ Create project folder

mkdir cricdash && cd cricdash

**3) Project Structure (copy/paste)**

cricdash/

├── app.py

├── pages/

│ ├── 1\_🏏\_Live.py

│ ├── 2\_📊\_Players.py

│ ├── 3\_🧮\_SQL\_Lab.py

│ └── 4\_🛠️\_Admin.py

├── services/

│ ├── cricbuzz\_api.py

│ ├── ingestion.py

│ └── transforms.py

├── utils/

│ ├── db\_connection.py

│ ├── settings.py

│ └── sql.py

├── models/

│ ├── schema.sql

│ └── orm.py # optional

├── assets/

│ └── logo.png (optional)

├── requirements.txt

└── .env.example

Create the files below as shown (you can paste them verbatim). Adjust URLs/keys to your provider.

**4) Configuration & Dependencies**

**requirements.txt**

streamlit>=1.36

requests>=2.32

sqlalchemy>=2.0

pydantic>=2.7

python-dotenv>=1.0

pandas>=2.2

tenacity>=8.4

alembic>=1.13

**.env.example**

# Database URL examples:

# Postgres: postgresql+psycopg2://user:pass@localhost:5432/cricdash

# MySQL: mysql+pymysql://user:pass@localhost:3306/cricdash

# SQLite: sqlite:///./cricdash.db

DATABASE\_URL=sqlite:///./cricdash.db

# REST API config (adjust to your source)

CRICBUZZ\_BASE\_URL=https://YOUR-CRICKET-API.com

CRICBUZZ\_API\_KEY=replace-me

# Polling & caching

POLL\_INTERVAL\_SECONDS=15

CACHE\_TTL\_SECONDS=10

# Development helpers

MOCK\_MODE=false

Copy .env.example to .env and fill in real values.

**5) Database Schema (DB‑agnostic SQL)**

**models/schema.sql**

-- Use generic SQL types for portability

CREATE TABLE IF NOT EXISTS teams (

team\_id VARCHAR(64) PRIMARY KEY,

name VARCHAR(255) NOT NULL,

short\_name VARCHAR(64),

country VARCHAR(128),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE IF NOT EXISTS players (

player\_id VARCHAR(64) PRIMARY KEY,

full\_name VARCHAR(255) NOT NULL,

batting\_style VARCHAR(64),

bowling\_style VARCHAR(64),

role VARCHAR(64),

country VARCHAR(128),

dob DATE,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE IF NOT EXISTS series (

series\_id VARCHAR(64) PRIMARY KEY,

name VARCHAR(255) NOT NULL,

season VARCHAR(64),

start\_date DATE,

end\_date DATE,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE IF NOT EXISTS matches (

match\_id VARCHAR(64) PRIMARY KEY,

series\_id VARCHAR(64),

team1\_id VARCHAR(64),

team2\_id VARCHAR(64),

venue VARCHAR(255),

start\_time TIMESTAMP,

status VARCHAR(64), -- scheduled / live / completed

result\_text VARCHAR(255),

toss VARCHAR(255),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY(series\_id) REFERENCES series(series\_id),

FOREIGN KEY(team1\_id) REFERENCES teams(team\_id),

FOREIGN KEY(team2\_id) REFERENCES teams(team\_id)

);

-- Minimal live score table (expand to ball-by-ball if needed)

CREATE TABLE IF NOT EXISTS innings (

innings\_id VARCHAR(64) PRIMARY KEY,

match\_id VARCHAR(64) NOT NULL,

batting\_team VARCHAR(64) NOT NULL,

runs INTEGER DEFAULT 0,

wickets INTEGER DEFAULT 0,

overs DECIMAL(5,1) DEFAULT 0,

cr\_runrate DECIMAL(5,2),

target INTEGER,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY(match\_id) REFERENCES matches(match\_id),

FOREIGN KEY(batting\_team) REFERENCES teams(team\_id)

);

-- Player aggregates per match (batting & bowling combined minimal)

CREATE TABLE IF NOT EXISTS player\_match\_stats (

match\_id VARCHAR(64) NOT NULL,

player\_id VARCHAR(64) NOT NULL,

team\_id VARCHAR(64),

role VARCHAR(64),

batting\_runs INTEGER,

batting\_balls INTEGER,

fours INTEGER,

sixes INTEGER,

sr DECIMAL(6,2),

bowling\_overs DECIMAL(5,1),

bowling\_runs INTEGER,

bowling\_wkts INTEGER,

econ DECIMAL(6,2),

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

PRIMARY KEY (match\_id, player\_id),

FOREIGN KEY(match\_id) REFERENCES matches(match\_id),

FOREIGN KEY(player\_id) REFERENCES players(player\_id)

);

CREATE INDEX IF NOT EXISTS idx\_matches\_status ON matches(status);

CREATE INDEX IF NOT EXISTS idx\_innings\_match ON innings(match\_id);

CREATE INDEX IF NOT EXISTS idx\_pms\_player ON player\_match\_stats(player\_id);

You can add ball‑by‑ball tables later for deeper analytics; start with the above for simplicity.

Apply the schema (SQLite applies automatically on first run, Postgres/MySQL require executing the SQL once).

**6) Core Utilities**

**utils/settings.py** — centralized config

from pydantic import BaseSettings, Field

class Settings(BaseSettings):

database\_url: str = Field("sqlite:///./cricdash.db", alias="DATABASE\_URL")

cricbuzz\_base\_url: str = Field(..., alias="CRICBUZZ\_BASE\_URL")

cricbuzz\_api\_key: str = Field("", alias="CRICBUZZ\_API\_KEY")

poll\_interval\_seconds: int = Field(15, alias="POLL\_INTERVAL\_SECONDS")

cache\_ttl\_seconds: int = Field(10, alias="CACHE\_TTL\_SECONDS")

mock\_mode: bool = Field(False, alias="MOCK\_MODE")

class Config:

env\_file = ".env"

case\_sensitive = False

settings = Settings()

**utils/db\_connection.py** — DB‑agnostic engine + helpers

from sqlalchemy import create\_engine, text

from sqlalchemy.engine import Engine

from contextlib import contextmanager

from typing import Any, Dict, Iterable, Optional

from utils.settings import settings

import pandas as pd

import threading

\_engine\_lock = threading.Lock()

\_engine: Optional[Engine] = None

def get\_engine() -> Engine:

global \_engine

if \_engine is None:

with \_engine\_lock:

if \_engine is None:

\_engine = create\_engine(

settings.database\_url,

pool\_pre\_ping=True,

pool\_size=5,

max\_overflow=10,

future=True,

)

return \_engine

@contextmanager

def connect():

engine = get\_engine()

with engine.begin() as conn:

yield conn

def fetch\_df(sql: str, params: Optional[Dict[str, Any]] = None) -> pd.DataFrame:

with connect() as conn:

return pd.read\_sql(text(sql), conn, params=params)

def execute(sql: str, params: Optional[Dict[str, Any]] = None) -> None:

with connect() as conn:

conn.execute(text(sql), params or {})

def upsert\_generic(table: str, key\_cols: Iterable[str], data: Dict[str, Any]):

"""DB‑agnostic upsert via SELECT -> UPDATE/INSERT inside a transaction.

Not as fast as native upsert, but portable.

"""

keys = list(key\_cols)

non\_keys = [k for k in data.keys() if k not in keys]

where\_clause = " AND ".join([f"{k} = :{k}" for k in keys])

set\_clause = ", ".join([f"{k} = :{k}" for k in non\_keys])

insert\_cols = ", ".join(data.keys())

insert\_vals = ", ".join([f":{k}" for k in data.keys()])

with connect() as conn:

exists = conn.execute(text(f"SELECT 1 FROM {table} WHERE {where\_clause} LIMIT 1"), data).scalar()

if exists:

if set\_clause:

conn.execute(text(f"UPDATE {table} SET {set\_clause}, updated\_at = CURRENT\_TIMESTAMP WHERE {where\_clause}"), data)

else:

conn.execute(text(f"INSERT INTO {table} ({insert\_cols}) VALUES ({insert\_vals})"), data)

**utils/sql.py** — small SQL helpers

from typing import Dict, Any, Optional

from utils.db\_connection import fetch\_df, execute

def run\_query(sql: str, params: Optional[Dict[str, Any]] = None):

return fetch\_df(sql, params)

def safe\_execute(sql: str, params: Optional[Dict[str, Any]] = None):

execute(sql, params)

**7) API Client & Transforms**

**services/cricbuzz\_api.py** — resilient REST client (requests)

import time

from typing import Dict, Optional

import requests

from requests.adapters import HTTPAdapter

from urllib3.util.retry import Retry

from utils.settings import settings

class CricbuzzClient:

def \_\_init\_\_(self, base\_url: Optional[str] = None, api\_key: Optional[str] = None):

self.base\_url = base\_url or settings.cricbuzz\_base\_url.rstrip("/")

self.api\_key = api\_key or settings.cricbuzz\_api\_key

self.session = requests.Session()

retries = Retry(

total=5,

backoff\_factor=0.4,

status\_forcelist=(429, 500, 502, 503, 504),

allowed\_methods=("GET",),

)

adapter = HTTPAdapter(max\_retries=retries)

self.session.mount("http://", adapter)

self.session.mount("https://", adapter)

self.headers = {

"User-Agent": "cricdash/1.0",

"Accept": "application/json",

}

if self.api\_key:

self.headers["Authorization"] = f"Bearer {self.api\_key}"

def \_get(self, path: str, params: Optional[Dict] = None):

url = f"{self.base\_url}/{path.lstrip('/')}"

r = self.session.get(url, headers=self.headers, params=params or {}, timeout=10)

r.raise\_for\_status()

return r.json()

# --- Endpoints (adjust names/paths to your provider) ---

def live\_matches(self) -> Dict:

return self.\_get("matches/live")

def match\_detail(self, match\_id: str) -> Dict:

return self.\_get(f"matches/{match\_id}")

def player\_detail(self, player\_id: str) -> Dict:

return self.\_get(f"players/{player\_id}")

def series\_detail(self, series\_id: str) -> Dict:

return self.\_get(f"series/{series\_id}")

**services/transforms.py** — normalize provider JSON → relational rows

from typing import Dict, Any, Iterable, List

# Note: Adjust keys to your provider's payload

def team\_row(api\_team: Dict[str, Any]) -> Dict[str, Any]:

return {

"team\_id": str(api\_team.get("id")),

"name": api\_team.get("name"),

"short\_name": api\_team.get("abbr"),

"country": api\_team.get("country"),

}

def player\_row(api\_player: Dict[str, Any]) -> Dict[str, Any]:

return {

"player\_id": str(api\_player.get("id")),

"full\_name": api\_player.get("name"),

"batting\_style": api\_player.get("battingStyle"),

"bowling\_style": api\_player.get("bowlingStyle"),

"role": api\_player.get("role"),

"country": api\_player.get("country"),

"dob": api\_player.get("dob"), # ensure ISO date in provider

}

def match\_row(api\_match: Dict[str, Any]) -> Dict[str, Any]:

return {

"match\_id": str(api\_match.get("id")),

"series\_id": str(api\_match.get("seriesId")) if api\_match.get("seriesId") else None,

"team1\_id": str(api\_match.get("team1", {}).get("id")),

"team2\_id": str(api\_match.get("team2", {}).get("id")),

"venue": api\_match.get("venue"),

"start\_time": api\_match.get("startTime"), # ISO 8601

"status": api\_match.get("status"),

"result\_text": api\_match.get("result"),

"toss": api\_match.get("toss"),

}

def innings\_rows(api\_match: Dict[str, Any]) -> List[Dict[str, Any]]:

rows = []

for inng in api\_match.get("innings", []):

rows.append({

"innings\_id": str(inng.get("id")),

"match\_id": str(api\_match.get("id")),

"batting\_team": str(inng.get("battingTeam", {}).get("id")),

"runs": inng.get("runs"),

"wickets": inng.get("wickets"),

"overs": inng.get("overs"),

"cr\_runrate": inng.get("runRate"),

"target": inng.get("target"),

})

return rows

def player\_match\_rows(api\_match: Dict[str, Any]) -> List[Dict[str, Any]]:

rows = []

for p in api\_match.get("players", []):

rows.append({

"match\_id": str(api\_match.get("id")),

"player\_id": str(p.get("id")),

"team\_id": str(p.get("teamId")),

"role": p.get("role"),

"batting\_runs": p.get("batting", {}).get("runs"),

"batting\_balls": p.get("batting", {}).get("balls"),

"fours": p.get("batting", {}).get("fours"),

"sixes": p.get("batting", {}).get("sixes"),

"sr": p.get("batting", {}).get("strikeRate"),

"bowling\_overs": p.get("bowling", {}).get("overs"),

"bowling\_runs": p.get("bowling", {}).get("runs"),

"bowling\_wkts": p.get("bowling", {}).get("wickets"),

"econ": p.get("bowling", {}).get("economy"),

})

return rows

**services/ingestion.py** — ingestion + upsert orchestration

from utils.db\_connection import upsert\_generic

from services.cricbuzz\_api import CricbuzzClient

from services.transforms import team\_row, player\_row, match\_row, innings\_rows, player\_match\_rows

from utils.db\_connection import connect

from typing import Dict

class IngestionService:

def \_\_init\_\_(self, client: CricbuzzClient):

self.client = client

def ingest\_match\_detail(self, match\_id: str):

m = self.client.match\_detail(match\_id)

# Teams

for t in [m.get("team1"), m.get("team2")]:

if t:

upsert\_generic("teams", ["team\_id"], team\_row(t))

# Match

upsert\_generic("matches", ["match\_id"], match\_row(m))

# Innings

for row in innings\_rows(m):

upsert\_generic("innings", ["innings\_id"], row)

# Players + per‑match stats

for row in player\_match\_rows(m):

upsert\_generic("player\_match\_stats", ["match\_id", "player\_id"], row)

# Ensure player master exists (may be partial; enrich via player\_detail later)

upsert\_generic("players", ["player\_id"], {

"player\_id": row["player\_id"],

"full\_name": row.get("player\_id"), # temporary placeholder

})

def refresh\_live(self):

data = self.client.live\_matches()

for m in data.get("matches", []):

match\_id = str(m.get("id"))

# lightweight upsert from list payload

upsert\_generic("matches", ["match\_id"], match\_row(m))

# optional: fetch full detail for live

self.ingest\_match\_detail(match\_id)

def enrich\_player(self, player\_id: str):

p = self.client.player\_detail(player\_id)

upsert\_generic("players", ["player\_id"], player\_row(p))

**8) Streamlit App — Pages & UI**

**app.py — global setup & homepage**

import streamlit as st

from utils.settings import settings

st.set\_page\_config(page\_title="CricDash", page\_icon="🏏", layout="wide")

st.title("🏏 CricDash — Cricket Analytics Dashboard")

st.markdown(

f"""

\*\*Data Source\*\*: REST API at `{settings.cricbuzz\_base\_url}`

\*\*DB\*\*: `{settings.database\_url}`

Polling every \*\*{settings.poll\_interval\_seconds}s\*\* with cache TTL \*\*{settings.cache\_ttl\_seconds}s\*\*.

Use the sidebar to navigate: \*\*Live\*\*, \*\*Players\*\*, \*\*SQL Lab\*\*, \*\*Admin\*\*.

"""

)

**pages/1\_🏏\_Live.py — real‑time match center**

import streamlit as st

import pandas as pd

from utils.settings import settings

from utils.db\_connection import fetch\_df

from services.cricbuzz\_api import CricbuzzClient

from services.ingestion import IngestionService

st.title("🏏 Live Matches")

# Auto‑refresh while viewing this page

st.autorefresh(interval=settings.poll\_interval\_seconds \* 1000, key="live\_refresh")

client = CricbuzzClient()

ingestor = IngestionService(client)

# Pull fresh data → DB (safe to run often)

ingestor.refresh\_live()

# List live/scheduled matches from DB

matches = fetch\_df("SELECT match\_id, status, venue, start\_time, team1\_id, team2\_id, result\_text FROM matches WHERE status IN ('live','scheduled') ORDER BY start\_time DESC")

if matches.empty:

st.info("No live or scheduled matches at the moment.")

else:

selected = st.selectbox("Select a match", matches["match\_id"].tolist())

# Pull detail (ensures innings & players are up‑to‑date)

ingestor.ingest\_match\_detail(selected)

# Scoreboard

inns = fetch\_df("""

SELECT i.innings\_id, t.name AS batting\_team, i.runs, i.wickets, i.overs, i.cr\_runrate, i.target

FROM innings i JOIN teams t ON t.team\_id = i.batting\_team

WHERE i.match\_id = :mid ORDER BY i.innings\_id

""", {"mid": selected})

st.subheader("Scorecard")

st.dataframe(inns, use\_container\_width=True, hide\_index=True)

# Player stats table (batting + bowling)

pms = fetch\_df("""

SELECT p.full\_name, pms.role, pms.batting\_runs, pms.batting\_balls, pms.fours, pms.sixes, pms.sr,

pms.bowling\_overs, pms.bowling\_runs, pms.bowling\_wkts, pms.econ

FROM player\_match\_stats pms JOIN players p ON p.player\_id = pms.player\_id

WHERE pms.match\_id = :mid

ORDER BY COALESCE(pms.batting\_runs,0) DESC, COALESCE(pms.bowling\_wkts,0) DESC

""", {"mid": selected})

with st.expander("Players in this match"):

st.dataframe(pms, use\_container\_width=True, hide\_index=True)

**pages/2\_📊\_Players.py — explorer & charts**

import streamlit as st

from utils.db\_connection import fetch\_df

st.title("📊 Player Explorer")

q = st.text\_input("Search player by name contains:", "")

if q:

df = fetch\_df("""

SELECT player\_id, full\_name, role, batting\_style, bowling\_style, country

FROM players

WHERE lower(full\_name) LIKE lower(:q)

ORDER BY full\_name

LIMIT 100

""", {"q": f"%{q}%"})

else:

df = fetch\_df("SELECT player\_id, full\_name, role, country FROM players ORDER BY updated\_at DESC LIMIT 100")

st.dataframe(df, use\_container\_width=True, hide\_index=True)

player\_id = st.text\_input("Enter Player ID to see aggregates")

if player\_id:

agg = fetch\_df("""

SELECT

SUM(COALESCE(batting\_runs,0)) as runs,

SUM(COALESCE(batting\_balls,0)) as balls,

SUM(COALESCE(fours,0)) as fours,

SUM(COALESCE(sixes,0)) as sixes,

ROUND(CASE WHEN SUM(COALESCE(batting\_balls,0))>0 THEN SUM(batting\_runs)\*100.0/SUM(batting\_balls) ELSE NULL END,2) as strike\_rate,

SUM(COALESCE(bowling\_wkts,0)) as wkts,

SUM(COALESCE(bowling\_runs,0)) as bowl\_runs

FROM player\_match\_stats WHERE player\_id = :pid

""", {"pid": player\_id})

st.subheader("Career summary (from stored matches)")

st.dataframe(agg, use\_container\_width=True, hide\_index=True)

**pages/3\_🧮\_SQL\_Lab.py — SQL console (read‑only by default)**

import streamlit as st

from utils.sql import run\_query

import pandas as pd

st.title("🧮 SQL Lab (Read‑only)")

st.caption("Run ad‑hoc SELECT queries against your database. LIMIT results to avoid heavy scans.")

sql = st.text\_area("SQL", "SELECT match\_id, status, start\_time, venue FROM matches ORDER BY start\_time DESC LIMIT 50;", height=150)

if st.button("Run"):

try:

df = run\_query(sql)

st.success(f"Returned {len(df)} rows")

st.dataframe(df, use\_container\_width=True)

st.download\_button("Download CSV", df.to\_csv(index=False).encode("utf-8"), file\_name="query.csv")

except Exception as e:

st.error(f"Error: {e}")

# Optional: toggle to allow non‑SELECTs for admins only

**pages/4\_🛠️\_Admin.py — CRUD forms**

import streamlit as st

from utils.db\_connection import execute, fetch\_df

st.title("🛠️ Admin — CRUD")

st.header("Add/Update Player")

with st.form("player\_form"):

player\_id = st.text\_input("Player ID", "")

full\_name = st.text\_input("Full Name", "")

role = st.text\_input("Role", "")

batting\_style = st.text\_input("Batting Style", "")

bowling\_style = st.text\_input("Bowling Style", "")

country = st.text\_input("Country", "")

submitted = st.form\_submit\_button("Save Player")

if submitted and player\_id:

# Upsert via generic pattern

from utils.db\_connection import upsert\_generic

upsert\_generic("players", ["player\_id"], {

"player\_id": player\_id,

"full\_name": full\_name or player\_id,

"role": role or None,

"batting\_style": batting\_style or None,

"bowling\_style": bowling\_style or None,

"country": country or None,

})

st.success(f"Player {player\_id} saved.")

st.divider()

st.header("Delete Player")

del\_id = st.text\_input("Player ID to delete", "")

if st.button("Delete") and del\_id:

try:

execute("DELETE FROM players WHERE player\_id = :pid", {"pid": del\_id})

st.success(f"Deleted {del\_id}")

except Exception as e:

st.error(str(e))

st.divider()

st.header("Browse Players")

st.dataframe(fetch\_df("SELECT \* FROM players ORDER BY updated\_at DESC LIMIT 100"), use\_container\_width=True)

**9) Running Locally (Step‑by‑Step)**

1. **Install deps**

pip install -r requirements.txt

1. **Create DB & apply schema**

* **SQLite**: nothing to install; first run will create cricdash.db automatically. To initialize tables now:

python - <<'PY'

from utils.db\_connection import execute

sql = open('models/schema.sql').read()

for stmt in [s for s in sql.split(';') if s.strip()]:

execute(stmt)

print('Schema applied')

PY

* **PostgreSQL/MySQL**: create a database, set DATABASE\_URL in .env, and run models/schema.sql with your client (psql, mysql).

1. **Configure .env**

cp .env.example .env

# edit .env with your DATABASE\_URL and API settings

1. **Start the app**

streamlit run app.py

1. **Navigate** to the **Live** page to see polling and scorecards. Use **Admin** for CRUD.

**10) Real‑Time Updates & Caching**

* The Live page calls ingestor.refresh\_live() each refresh cycle.
* st.autorefresh(interval=...ms) ensures the page re‑runs every N seconds.
* You can add @st.cache\_data(ttl=settings.cache\_ttl\_seconds) on heavy read queries to reduce DB churn.
* For background sync, you can run a separate **cron/Streamlit script** that periodically calls refresh\_live(); keep the UI responsive.

**Example cached query wrapper**

import streamlit as st

from utils.db\_connection import fetch\_df

@st.cache\_data(ttl=15)

def latest\_live\_matches():

return fetch\_df("SELECT \* FROM matches WHERE status='live' ORDER BY start\_time DESC")

**11) SQL‑Driven Analytics Examples**

* **Top batters by strike rate (min 200 balls)**

SELECT p.full\_name,

SUM(pms.batting\_runs) AS runs,

SUM(pms.batting\_balls) AS balls,

ROUND(SUM(pms.batting\_runs)\*100.0 / NULLIF(SUM(pms.batting\_balls),0), 2) AS SR

FROM player\_match\_stats pms

JOIN players p ON p.player\_id = pms.player\_id

GROUP BY p.full\_name

HAVING SUM(pms.batting\_balls) >= 200

ORDER BY SR DESC

LIMIT 20;

* **Best bowling (min 30 overs)**

SELECT p.full\_name,

SUM(pms.bowling\_wkts) AS wkts,

ROUND(SUM(pms.bowling\_runs) / NULLIF(SUM(pms.bowling\_overs),0), 2) AS Econ

FROM player\_match\_stats pms

JOIN players p ON p.player\_id = pms.player\_id

GROUP BY p.full\_name

HAVING SUM(pms.bowling\_overs) >= 30

ORDER BY wkts DESC, Econ ASC

LIMIT 20;

* **Chasing performance** (requires target in innings)

SELECT t.name AS team, COUNT(\*) AS chases,

SUM(CASE WHEN i.runs >= i.target AND i.target IS NOT NULL THEN 1 ELSE 0 END) AS successful

FROM innings i JOIN teams t ON t.team\_id = i.batting\_team

GROUP BY t.name

ORDER BY successful DESC;

**12) CRUD Patterns (Explained)**

* **Generic upsert** used here is portable and safe, but slower than vendor‑specific UPSERT. For high volume, switch to dialect‑specific statements:
  + PostgreSQL: INSERT ... ON CONFLICT (key) DO UPDATE
  + MySQL: INSERT ... ON DUPLICATE KEY UPDATE
  + SQLite: INSERT OR REPLACE or INSERT ... ON CONFLICT (modern SQLite)
* Keep **admin forms** minimal but validate IDs and types. Prefer **server‑side validation** (e.g., check player\_id exists before delete).

**13) Error Handling, Retries, & Rate Limits**

* CricbuzzClient enables robust retries on 429/5xx with exponential backoff.
* Always use **timeouts** (done in \_get with timeout=10).
* Consider **request throttling** (e.g., sleep between calls, or a ratelimit decorator) for providers with strict QPS limits.

**14) Security & Secrets**

* Never hard‑code API keys; load from .env / environment variables.
* Use **read‑only** SQL Lab for general users. Gate mutations behind an admin toggle or auth layer.
* If deploying publicly, add auth (e.g., Streamlit Community auth, custom OAuth proxy, or a reverse proxy). Avoid exposing raw SQL write access.

**15) Deployment Tips**

* **SQLite** is fine for a single‑user MVP. For multi‑user/prod: **PostgreSQL**.
* Containerize with Docker; map in .env, expose port 8501, link a DB service.
* Add **Alembic** migrations for schema evolution.

**Dockerfile (optional)**

FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8501

CMD ["streamlit", "run", "app.py", "--server.port=8501", "--server.address=0.0.0.0"]

**docker-compose.yml (optional)**

version: "3.9"

services:

db:

image: postgres:16

environment:

POSTGRES\_USER: cric

POSTGRES\_PASSWORD: cric

POSTGRES\_DB: cricdash

ports: ["5432:5432"]

app:

build: .

environment:

DATABASE\_URL: postgresql+psycopg2://cric:cric@db:5432/cricdash

CRICBUZZ\_BASE\_URL: ${CRICBUZZ\_BASE\_URL}

CRICBUZZ\_API\_KEY: ${CRICBUZZ\_API\_KEY}

ports: ["8501:8501"]

depends\_on: [db]

**16) Troubleshooting Checklist**

* **No data appears**: Verify API URL/key and that endpoints return JSON with fields expected by transforms.py.
* **DB errors**: Check DATABASE\_URL driver (e.g., psycopg2 for Postgres, pymysql for MySQL). Install missing driver.
* **Live page not updating**: Increase POLL\_INTERVAL\_SECONDS; confirm ingestor.refresh\_live() is called; check logs for HTTP errors.
* **Wrong columns**: Align your provider’s JSON keys with mapping functions in transforms.py.

**17) Nice‑to‑Haves (Next Iteration)**

* Ball‑by‑ball ingestion and win‑probability graphs
* Team/Player head‑to‑head dashboards
* Materialized views for fast aggregates
* Role‑based auth and audit logging for CRUD changes
* Scheduled background worker for near‑real‑time sync

**18) Summary — What you’ll have running**

* A **Streamlit** app with pages for **Live**, **Players**, **SQL Lab**, and **Admin**
* A robust **REST client** with retries and timeouts
* A clean **relational schema** with indexes
* **DB‑agnostic** code via **SQLAlchemy**
* A **generic UPSERT** strategy and form‑based **CRUD**

Paste the provided files, configure .env, apply schema.sql, and run streamlit run app.py. As you wire your actual API payload, update transforms.py field names accordingly.