Adam Nash

CUS-720-0

Specification Phase

10/10/2025

NashBoard

Introduction

NashBoard is a sports analytics platform designed for everyday fans who love the game but aren't fluent in advanced statistics. Its purpose is simple: take the complicated numbers that drive modern sports and make them understandable, visual, and useful. The service brings together real-time data from professional sports APIs, transforming it into a clear, interactive dashboard that lets users learn and explore how their favorite games work at a deeper level. The platform targets uneducated or casual sports fans who want to follow their teams and players more intelligently without needing a background in analytics. By showing stats in plain language and giving users the power to personalize what they see, NashBoard bridges the gap between raw data and actual understanding. The domain of this system falls within sports data analytics and fan engagement technology, blending web development, data science, and visualization into a single streamlined product.

Project Scope

NashBoard focuses on making sports data easily available and ready for the user to easily understand by providing real-time stats, player and team comparisons, and educational explanations. Its primary features include live dashboards, customizable layouts, and glossary-style stat breakdowns.

Out of Scope

While future versions of NashBoard may integrate a machine learning-based recommendation system to identify underperforming players on a user's fantasy team, this functionality is not part of the current scope. The reasoning behind this is that for this phase, the focus is on real-time data visualization, stat explanation, and matchup analysis. Any recommendation or ranking system will be considered an extension beyond the current specification.

User functional requirements

This section gives a detailed outline of the main actions a user can perform on Nashboard. Each requirement has a focus on making sure the platform is easy to use as well as accessible to non-technical fans. The goal is to give users primary control over how they use the website as they explore and learn from the data.

ID	Requirement (The system shall…)	Trigger / Inputs	System Response	Notes (Pre/Postconditions, Rationale)
UFR-01	allow users to create an account with an email and a password.	User submits registration form.	Account record is created and stored securely.	Creating an account is the first step that lets the system actually recognize someone and give them their own personalized space. This means the platform has to store emails and passwords safely, but also handle the little things like catching typos or duplicate accounts without confusing the user. Developers should make the sign-up flow feel quick and not overly technical, since most people just want to get inside the app without thinking too much about security details. Even though hashing passwords is a behind-the-scenes thing, the reliability of this process matters a lot because everything else—favorites, layouts, saved modes—depends on it. The whole point is to make users feel like they're creating *their own version* of NashBoard, not just logging into some generic stats website.

UFR-02	allow users to reset passwords via email.	User requests a password reset.	System sends a secure reset link to the user's email.	People forget passwords all the time, so this needs to be simple and not make users panic. The system should quietly generate a secure link and send it by email, and the link shouldn't last forever because that would be a security issue. It's important that the reset page looks reassuring and straightforward so even someone who isn't tech-savvy can finish the process without second-guessing themselves. Developers also need to make sure old or reused links don't work, since that could open the door to unwanted access. A smooth reset flow builds trust, and trust is something users notice even if they don't talk about it.
UFR-03	Let users select a sport and view the corresponding dashboard.	User selects a sport (e.g., NFL, NBA).	Dashboard updates to show widgets and data for the chosen sport.	Switching sports should feel almost instant, like flipping channels on TV, and the dashboard should refresh cleanly without leftover bits from the previous sport. The system has to be careful about loading widgets that only fit the chosen sport, otherwise the interface gets messy or confusing. This transition also needs to stay smooth for casual fans who may not know all the menu options, so the UI should guide them naturally. Good caching helps here because

				users might bounce between sports, especially during busy weekends. It's a small feature on the surface but sets the tone for how "fast" and responsive NashBoard feels as a whole.
UFR-04	display live scores, stats, and injuries for ongoing games.	User is on a sports dashboard while games are active.	System shows updated game tiles with core stats/injury info.	People mainly check sports apps during live games, so this part has to feel reliable even when everyone else is checking scores at the same time. The platform must pull fresh data frequently, but not so aggressively that it overwhelms the system or the external APIs. Injuries, in particular, need to stand out clearly because they affect how users read the rest of the stats. When something goes wrong with the data source—and it will at some point—the app should show fallback info or a friendly message instead of a blank widget. Users should always know whether they're seeing live info, a slightly delayed update, or temporary placeholder data. These small touches make the app feel alive and trustworthy.

UFR-05	Let users customize the dashboard layout by adding, removing, and arranging widgets.	User drags, adds, or removes widgets.	The layout is updated immediately and saved to the user profile.	A customizable dashboard is what makes NashBoard feel like *your* space, not just another stats feed. The dragging, dropping, and saving of widgets should feel natural and not awkward or glitchy. Whenever the user changes their layout, the system should save it right away so nothing gets lost if they refresh the page or switch devices. Beginners shouldn't feel intimidated by having too many customization options, but advanced users shouldn't feel restricted either, so the design needs a careful balance. Developers should expect that users will add and remove widgets frequently, so the layout engine needs to handle lots of small changes without breaking. This gives the platform a sense of ownership that keeps people coming back.
UFR-06	Provide a search for teams, players, and games.	User enters search text or filters.	System returns a list of matching entities.	Search is one of those features people take for granted until it doesn't work, so it needs to feel natural and fast. Users shouldn't have to type things perfectly to get results, which means the system has to handle partial spellings, abbreviations, and sometimes even mistakes. The results should show up quickly so users don't feel like

				they’re waiting around just to find something simple. Developers also need to think about how to separate players, teams, and games in a way that doesn’t confuse beginners. A clean search feature makes the whole platform feel smarter and easier to navigate. It’s one of the big things that helps users actually “find their way” around NashBoard.
UFR-07	allow users to compare teams or players side-by-side.	User selects two or more entities to compare.	The system displays a comparison view with key stats.	Comparing things is something sports fans naturally love doing, so the system should make this feel easy and kind of fun. The comparison view needs to use consistent stats so users don’t wonder why things don’t line up the same way between two players. In Beginner Mode, the comparison should basically point out the big differences without overwhelming the user, but in Advanced Mode it should let serious users go deeper. Developers should aim for a layout that stays organized even when comparing more detailed stats. The experience should help users learn how to interpret stats without them even realizing they’re learning. A good comparison view makes the app feel like a

				place for real insights, not just raw numbers.
UFR-08	Let users mark favorite teams and players.	User clicks “favorite” on a team/player.	System stores favorites and highlights them in dashboards.	Letting users favorite teams or players makes the app feel more personal right away. This feature needs to be effortless — ideally just a single tap — and the system should update instantly so it feels responsive. Favorites help the dashboard show the most relevant information without the user having to search every time. Developers also need to remember that favorites are something users expect to sync across devices, so saving them properly matters a lot. As the platform grows, favorites could unlock more features, so building it cleanly now will pay off later. The main goal is to make users feel like NashBoard is tailored to their interests from the moment they log in.
UFR-09	allow toggle between light and dark mode.	User switches theme in settings or UI.	UI theme changes immediately, and preference is saved.	Switching themes seems like a small feature, but users notice when it’s missing or poorly implemented. The toggle should respond instantly, and nothing in the UI should flicker or look out of place during the transition.

				Developers also need to check that charts and widgets remain readable in both themes, since some colors might only work in one mode. Storing the user's theme preference is important so the system doesn't switch back unexpectedly. Even though this feels like a cosmetic detail, it plays a big part in whether the interface feels comfortable to use for long periods. It also signals that the app is built with real user habits in mind.
UFR-10	Support exporting selected data as PDF or CSV.	User clicks export on a widget/dash board.	The system generates and returns a downloadable file.	Exporting data gives users a sense of ownership over the information they're exploring, and it can be surprisingly useful for fantasy players, students, or analysts. A PDF export should look neat and readable, while a CSV export should focus on making the numbers easy to work with in another tool. Developers must make sure the export matches whatever filters or mode the user currently has selected, otherwise the user might think the system is inaccurate. Handling large exports gracefully is important because some users will inevitably try to export everything at once. Even though exporting is optional, doing it well makes the platform feel more complete

				and professional. Users appreciate having the option even if they don't use it often.
UFR-11	support filtering data by date, team, season, and league.	User applies filters in UI.	Views update to reflect selected filters.	Filters help users slice the data in ways that match exactly what they're trying to understand, so the system needs to apply them consistently throughout a dashboard. Developers should make sure filters don't accidentally "fight each other," like when a date filter hides a team the user selected. It's important that filter selections are easy to see and change, especially for beginners who may forget what they clicked. The filtering system must be fast so the interface doesn't feel sluggish when someone applies multiple filters at once. This requirement also opens the door for deeper analysis because filters let users drill into specific situations. Overall, well-designed filters make the data feel more interactive and alive.
UFR-12	allow toggling between basic and advanced stat views.	User toggles view mode on a widget or dashboard.	System switches between minimal and detailed data views.	This requirement helps bridge the gap between casual fans and more experienced users without splitting the app into two separate experiences. The system needs to keep both versions of a widget ready to load so switching

				modes feels almost instant. Developers should make sure the layout doesn't shift around too much when switching views, because sudden jumps can feel jarring. Advanced Mode should reveal deeper numbers, but it shouldn't drown users in stats that have no explanation. Over time, users might move from Beginner to Advanced as they get more comfortable, so the transition should feel natural. This requirement gives NashBoard a flexibility that most sports sites don't have.
UFR-13	show leaderboards and trending players for a sport.	User opens a leaderboard /trending widget or section.	System displays ranked lists based on selected criteria.	Leaderboards give users a quick way to see who's doing well without having to dig around the app, so they need to feel fast and reliable. The rankings should make sense at a glance, even if a user isn't totally familiar with all the stats behind them. Developers have to be careful about how often they refresh these lists since sports data can spike during busy times, and users expect changes to reflect quickly but not jump around randomly. Showing trends, like players who are rising or slipping, adds a layer of insight that makes the platform feel smarter than a basic scoreboard. These lists should also look clean and readable, even on smaller

				screens. The overall goal is to help users get a “pulse check” on the league in just a few seconds.
UFR-14	show glossary definitions for stats and terms.	User hovers/clicks on a stat with glossary support.	Tooltip or modal shows a plain-language definition.	Many fans hear advanced stats on TV or social media but don’t fully understand what they mean, so the glossary plays a big role in making the app feel welcoming. Developers should aim for definitions that feel friendly and practical instead of overly technical or math-heavy. It’s important that these explanations appear right when the user needs them, not buried in a separate help page. Even small examples, like “This stat matters because…,” can make a huge difference for clarity. Glossary items should also be easy to update, since new analytics terms pop up every season. This requirement helps users learn gradually without ever feeling judged or overwhelmed.
UFR-15	show performance trends over time.	User opens trend view for a player/team or widget.	System displays time-series charts (e.g., last 10 games).	Trends help users understand whether a performance is a one-off event or part of a larger pattern, which is something every sports fan naturally wonders. Developers need to design charts that are easy to read and not overloaded with

				details, especially for beginners. Pulling historical data has to be efficient so the app doesn't lag when a user wants to see long time spans like multiple seasons. In Advanced Mode, trends can show deeper splits or filters, but they still should be tied to real-world meaning. Even casual users like noticing streaks or declines, so the visuals should highlight interesting changes clearly. This feature adds a lot of storytelling power to the stats.
UFR-16	Send daily or scheduled notifications (optional).	User opts in and configures preferences.	System sends notifications (e.g., upcoming games, milestones).	Notifications can keep users engaged, but they need to be relevant so they don’t feel spammy or random. The system should let users choose exactly what they want to be notified about, whether it’s game start times, injuries, or big performances. Developers also need to handle the timing carefully, since sending too many alerts at odd hours might annoy people. It’s important that each notification feels helpful, not like filler. This feature can eventually tie into favorites so users get updates tailored to the teams and players they care about most. Done well, notifications make the app feel more like a personal assistant for sports.

UFR-17	allow users to share widgets or dashboards via link (optional).	User clicks “Share” on a widget or dashboard.	The system generates a secure shareable link.	Sharing gives users a way to show interesting stats to friends, which can spread the platform naturally through word of mouth. The system needs to create links that show the exact state of the widget or dashboard at the moment it’s shared, so there’s no confusion later. Developers must make sure the shared version doesn’t expose anything private, like the user’s account info or settings. The shared page should still look clean even if the viewer isn’t logged in. This requirement helps NashBoard feel social without needing a full social network built in. It also encourages users to explore more because they know they can easily share cool findings.
UFR-18	allow users to access past data and previous stat versions.	User selects past dates or history mode.	The system shows historical data snapshots/versions.	Some fans like comparing current performances to previous seasons, so access to historical data makes the platform feel deeper and more trustworthy. Developers should keep older data separated in a way that avoids mixing it with live stats, which could confuse users. It’s important for the system to store “versions” because sports APIs sometimes change data after the fact. Users should always know

				whether they're looking at live data or a past snapshot. This requirement makes long-term analysis possible and gives the platform more credibility. It also gives advanced users a richer sandbox to explore trends or compare eras.
UFR-19	offer a Beginner Mode with simplified layouts.	User enables Beginner Mode in settings or UI.	System loads dashboard with simplified widgets/labels.	Beginner Mode is meant to welcome users who might feel intimidated by too many stats, so the system needs to feel calm and simple in this mode. Widgets should focus on the main ideas rather than dozens of numbers that might confuse newer fans. Developers should aim for an uncluttered layout that feels friendly and easy to follow, with larger text and fewer elements per screen. This mode should help users build confidence over time so they feel comfortable exploring Advanced Mode later. Saving the mode preference is important so users don't keep toggling it every time they log in. Beginner Mode helps widen the app's audience without sacrificing depth.
UFR-20	explain how advanced stats relate to game pace and flow.	User interacts with advanced stat widgets.	The system provides text explanations and simple visuals.	A lot of advanced stats only make sense when fans understand the real-game situations behind them, so explanations need to connect the numbers to what actually

				happens on the field or court. Developers should place these explanations close to the stats they describe so users don't need to hunt for context. Even short, simple sentences about why something matters can help users grow their understanding. The explanations should avoid heavy math or jargon that might turn beginners away. By doing this, the system acts almost like a teacher, helping users interpret metrics they might have ignored otherwise. Overall, this requirement gives life to the numbers and makes them easier to appreciate.
UFR-21	Show **Top 5 recommendations** in Beginner Mode for key widgets.	User views a widget in Beginner Mode.	System lists up to 5 ranked options (e.g., best matchups, top players) for the chosen stat.	Top 5 lists give beginners an easy place to start by showing what's immediately important without forcing them to sort through a lot of data. Developers will need to decide what criteria drive these rankings, and even if the logic is simple, it should still make sense to users. These recommendations should update naturally as games progress or new data comes in. The system should also explain why something made the list so it doesn't feel random. This feature keeps Beginner Mode helpful without overwhelming users.

				It gives people a clear sense of what to pay attention to during busy sports days.
UFR-22	Provide a **Widget Library** where users can browse widgets by sport and category, then add them to their dashboard.	User opens Widget Library in sidebar.	System displays a categorized widget list and adds selected widgets to the dashboard on request.	The Widget Library lets users discover new ways to see their data, so browsing it should feel smooth and not overloaded. Developers should group widgets in a way that feels logical to sports fans, like separating live-game widgets from historical-analysis ones. Widget previews should load quickly so users can make decisions without waiting or guessing. Adding a widget should immediately update the dashboard so the user gets instant confirmation. The library also needs to feel organized enough that new widgets can be added without breaking the layout. This requirement helps the app grow without confusing its users.
UFR-23	Provide a **Player Library** that supports filtering by sport, team, season, jersey number, and name, and shows a detailed player page.	User opens Player Library and applies filters/search.	System returns matching players and opens a detailed stats view on selection.	The Player Library gives fans an organized place to explore players across seasons, which is helpful for both beginners and seasoned analysts. Developers should make the filters work smoothly together so users

				can narrow things down quickly. The detailed player pages should change based on mode—Beginners see the basics, while advanced users get deeper stats and historical context. Injury history, trends, and comparisons should fit naturally into the page layout. This feature becomes especially helpful when users are following multiple teams or exploring unfamiliar players. Clean data management is key here because mistakes can easily confuse users.
UFR-24	Provide **Predictive Widgets** (e.g., passing yards, score projections) for selected games with Beginner and Advanced views.	User adds a predictive widget for a specific game.	System computes and shows projections (simple in Beginner Mode, detailed in Advanced).	Predictive widgets help fans look ahead instead of just reviewing what already happened, which makes the app feel more dynamic. Developers don't need full-blown machine learning for this phase, but the projections still need to feel grounded in recent performance and matchup context. Beginner Mode should show a simple projection range with a short explanation, while Advanced Mode can include breakdowns like efficiency splits or situational trends. These widgets should update automatically as new data or injuries come in during the week. Even if predictions aren't perfect, the clarity behind them is what builds

				trust. This requirement sets NashBoard apart from more basic score apps by offering forward-thinking insights.

System functional requirements

The following system functional requirements are definitive of the core technical capabilities of NashBoard. They attempt to focus on handling real-time data, ensuring secure connections, and keeping the backend fast, stable, and ready for future expansion.

ID	Requirement (The system shall…)	Key Inputs	Key Outputs	Notes / Rationale
SFR-01	store user information securely in a database.	Auth data, preferences.	Persisted user records.	The system needs to store user information in a way that developers can rely on later when loading preferences, layouts, or favorites. This means passwords can't just be saved as plain text — they must be hashed using something modern and safe so even if someone got access to the database, they still couldn't read them. Developers should also think about how profile settings are organized so the data is easy to query and update. If the database structure is messy, everything that depends on it becomes harder to maintain. This requirement basically forms the backbone for

				anything personalized in the app. A secure and clean user data layer protects both the users and the system long-term.
SFR-02	connect to third-party sports APIs to fetch data.	API keys, requests.	JSON feeds from providers.	Since NashBoard doesn't generate its own raw sports data, the app must reliably connect to external sports APIs to pull in live scores, player stats, injuries, and schedules. Developers will need to handle authentication, rate limits, and network hiccups because these APIs won't always behave perfectly. Each API may return data in slightly different shapes, so the system also needs a consistent method of requesting and receiving information. The reliability of this connection directly affects how "live" the live widgets feel to users. Once this pipeline is stable, developers can build much richer features on top of it. In short, this is the data lifeline for the entire platform.
SFR-03	normalize and clean incoming data from APIs.	Raw feeds.	Cleaned, normalized records.	Raw API data can be messy, inconsistent, or incomplete, so the system has to clean and normalize it before anything gets shown on the dashboard. Developers need

				a process for transforming different API formats into a single set of internal structures so the rest of the system doesn't have to worry about provider differences. If this step is skipped or done poorly, widgets will produce confusing results, like mismatched player names or missing stats. Normalization also makes comparison views and historical charts much easier to implement. This requirement sets the tone for the system's accuracy and reliability. A strong normalization layer saves countless headaches later on.
SFR-04	refresh live game data automatically at defined intervals.	Scheduler events.	Updated in-memory/cache & DB entries.	The system needs a consistent way to refresh live data so that users don't have to reload the page to see new scores. Developers should design a scheduler or polling mechanism that balances freshness with performance — updating too often can overload the system or hit API limits, while updating too slowly makes the app feel stale. When multiple games are happening at once, the refresh logic needs to handle simultaneous requests efficiently. Ideally, the system should also detect when a game ends so it can switch to

				a slower refresh rate. This requirement keeps the app feeling alive during game days. It also plays a big role in meeting user expectations for “real-time” information.
SFR-05	ProvideRESTful endpoints for frontend access.	HTTP requests.	JSON responses.	The frontend can’t directly access raw data sources, so the backend needs clean, well-structured REST endpoints that deliver exactly what the widgets need. Developers should design predictable routes and responses so the frontend team doesn’t have to guess where specific data comes from. These endpoints should handle errors gracefully instead of crashing the UI with broken JSON. If the endpoints are fast and reliable, the frontend will feel smooth and responsive. Good API design also makes it easier to add new widgets or sports later without rewriting everything. Essentially, this requirement defines the contract between the frontend and backend.
SFR-06	log API and user activity.	Events (API calls, actions).	Log entries.	Logging activity helps developers understand what’s happening inside the system, especially when something goes wrong. The system should record important events like API

				failures, login attempts, and unusual traffic patterns, but without flooding the logs with useless noise. These logs become crucial when debugging issues that only appear under certain conditions or specific user actions. Developers should also store logs in a place where they can be searched and analyzed easily, rather than scattered across random locations. Clear logging makes maintenance far easier, especially as the app grows. It also helps catch bugs before users notice them.
SFR-07	Maintain caching for faster data retrieval.	Repeated queries.	Cached responses.	Caching helps the system avoid repeatedly hitting external APIs or slow database queries, which keeps the app feeling fast even during busy sports moments. Developers need to decide which data should be cached and for how long, since some stats change constantly while others barely change at all. A smart caching strategy reduces both API costs and backend load. If implemented poorly, though, the system might show outdated data or fail to update during critical moments, so developers should be careful. Cache invalidation — knowing when

				to refresh or clear the cache — is just as important as caching itself. Overall, this requirement is key for performance and scalability.
SFR-08	Back up application data regularly.	Backup jobs.	Backup files.	Backups are something users never see, but they matter a lot when something unexpectedly goes wrong. The system should run backups automatically on a set schedule so developers don't have to remember to run them manually. These backups should include not just user accounts but also saved layouts, favorites, and historical data snapshots. If a system failure or accidental deletion happens, a clean backup is the only thing that prevents permanent damage. Developers also need a clear recovery process so restoring data isn't chaotic or slow. This requirement ensures the app can survive worst-case scenarios without losing user trust.
SFR-09	Send verification and password reset emails.	Email, tokens.	Email messages with secure links.	Email services sound simple, but they require careful setup so messages actually reach users instead of landing in spam. The system must generate secure tokens and send them through a trusted provider that supports encryption and modern

				authentication. Developers should also give the emails a clean, recognizable design so users know they're legitimate. If this system breaks, users can't verify accounts or recover access, which would make the platform frustrating to use. Handling email failures gracefully — such as retrying or showing friendly error messages — is also important. This requirement supports the reliability of the entire account system.
SFR-10	support multiple sports API integrations.	Multiple API configs.	Combined data views.	Because NashBoard aims to cover multiple sports, the system needs to be built with flexibility in mind. Each sport may require a different API with its own quirks, so the backend must support adding new integrations without major redesigns. Developers should use a modular approach so that NFL logic doesn't get tangled up with NBA logic, for example. This separation also helps isolate bugs since issues in one sport shouldn't affect another. Supporting multiple APIs also opens the door to expansions in future semesters or versions. This requirement keeps the platform adaptable instead of locked into one sport.

SFR-11	handle rate limiting and fallback behavior when APIs are overloaded or down.	API responses/errors.	Fallback responses, graceful messages.	External sports APIs often get hammered during big games, so the system needs a smart way to handle rate limits without everything breaking. Developers should expect that requests will sometimes be rejected or slowed down and design fallback strategies accordingly. This could mean showing cached data for a short period or reducing how frequently the platform tries to fetch updates. The goal is to avoid showing blank widgets or crashing the dashboard just because the data source is busy. Users should still get helpful information even when the live feed is struggling. This requirement helps maintain stability during peak sports moments when users rely on the app the most.
SFR-12	Maintain version control for computed metrics and data snapshots.	Metric updates.	Version records and history.	Because sports data can change after the fact or be recalculated by providers, the system needs a reliable versioning approach for storing metrics. This allows the app to show older stats exactly as they were originally recorded, which is important for trust and historical analysis. Developers should set up a way to tag snapshots so the

				backend always knows which version belongs to which date or game. Without a system like this, historical widgets could show confusing or inconsistent results. Versioning also makes debugging easier when users report differences between past and present calculations. This requirement is key for long-term accuracy and transparency.
SFR-13	Provide an admin panel to monitor data health.	Admin login.	Admin dashboard.	An admin panel gives developers or instructors a way to check the system's overall status without digging through logs or code. It should show whether API connections are working, how often data updates are succeeding, and if any services are failing silently. This panel doesn't need to be fancy, but it should be organized enough that someone unfamiliar with the backend can still make sense of it. Being able to quickly spot unusual patterns can prevent small issues from growing into bigger outages. It also helps instructors track system behavior during grading or demonstrations. This requirement makes ongoing maintenance more manageable and transparent.

SFR-14	support multi-device synchronization of user preferences and layouts.	Auth session.	Consistent state across devices.	Users expect their personalized dashboard to look the same no matter where they log in, whether that's a laptop, tablet, or phone. Developers need to store layouts, theme preferences, and favorite settings in a way that loads reliably across devices. The system should avoid situations where the layout appears differently depending on screen size unless the user intentionally adjusts it. Cloud storage or database-based syncing ensures the interface stays consistent and predictable. This requirement helps make the platform feel modern and well-integrated. Consistency across devices also encourages users to rely on NashBoard as a daily tool.
SFR-15	Run scheduled ingestion jobs for historical data and notifications.	Scheduler.	Loaded data, triggered notifications.	The backend needs to handle scheduled tasks like loading historical data or sending daily notifications, and these jobs need to run even if no one is actively using the system. Developers should make sure the scheduler they choose is stable and can recover from failures without manual intervention. These jobs might run during low-traffic hours to avoid stressing the system. If anything goes

				wrong during the ingestion process, the system should log it and retry gracefully. Having automated jobs like this keeps the data fresh without requiring constant developer attention. This requirement helps maintain the platform's reliability behind the scenes.
SFR-16	store data transformation metadata.	Processing events.	Transformation/lineage records.	Whenever raw sports data is processed or transformed, the system should record what was changed, when, and why. This helps developers trace unexpected results back to the original source if something seems off in a widget or chart. Metadata can also reveal when inconsistencies were introduced by external APIs or internal logic. Having a clear history of transformations is especially useful when debugging comparisons or predictive widgets. Even though most users never see this information, it supports the platform's credibility. This requirement strengthens the app's ability to explain or justify the data it presents.
SFR-17	manage secure tokens and sessions.	Tokens, cookies.	Validated sessions, secure storage.	Session management is essential for keeping user accounts safe, especially if the app is used on shared

				devices. Developers need to design a token system that prevents unauthorized access while keeping the login experience smooth. Tokens should expire after a reasonable amount of time or when suspicious activity is detected. The system must also protect against session hijacking by using secure cookies and enforcing strict domain rules. A reliable session layer helps users feel comfortable storing personal preferences in the app. This requirement ensures the foundation of account security remains strong.
SFR-18	Export data to files (PDF/CSV) upon request.	Export request.	File or link.	The backend must prepare exports in a format that feels professional and readable, which can require extra formatting work. Developers should make sure the export reflects whatever filters or settings the user selected so that the file matches what's on their screen. CSV exports are usually straightforward, but PDFs might need careful styling to avoid looking cluttered. The system also needs to handle errors gracefully — for example, if the user tries to export too much data at once. Speed matters too; users shouldn't feel like exporting locks up the system. This requirement

				supports flexible data usage beyond the platform.
SFR-19	Detect and handle API failures gracefully.	API call results.	User-friendly error messages and fallback data.	External APIs will fail sometimes, whether from heavy traffic, outages, or bad data responses. The system needs to recognize these situations quickly and prevent them from breaking the UI. Developers should design fallback paths, such as showing cached data or temporary placeholders, so the user still gets something useful. Clear messaging helps set expectations without alarming users. If errors aren't handled well, the platform can feel unreliable even if the underlying issue came from a third-party provider. This requirement helps maintain a smooth experience even during unpredictable moments.
SFR-20	support future machine learning integrations for analytics.	Model inputs/outputs.	Prediction results.	Even though full ML models aren't required right now, the system should be built with enough flexibility to plug in predictive components later. Developers should organize the backend so that projections, trends, and derived metrics can be updated without rewriting the whole system. This might

				mean creating a dedicated service or module where ML logic can eventually live. Planning for this early prevents major architectural headaches in future versions. It also allows students or instructors to experiment with model outputs down the road. This requirement keeps NashBoard open to growth and experimentation.
SFR-21	compute derived and predictive metrics for supported widgets and expose them via API.	Normalized data, game context, simple rules.	Derived stats, projections (e.g.,, passing yards, score).	Derived stats and simple projections help users make sense of performances beyond raw numbers. Developers need to create formulas or rules that produce meaningful insights without requiring full machine learning models. These calculations should be consistent so that widgets don't contradict each other. As new data comes in — like injuries or recent games — the projections should update smoothly. This requirement also supports the Top 5 features in Beginner Mode and more detailed breakdowns in Advanced Mode. It makes the app feel more intelligent and forward-looking than a basic scoreboard.

SFR-22	The system shall maintain a structured metadata service that defines each widget's properties, required data inputs, display modes, and supported sports.	Widget ID, Sport type, Widget configuration fields, Supported display mode	Widget metadata object, Render instructions for front end, list of available dodgers grouped by sport	This requirement ensures the backend can supply the frontend with all the information it needs to render widgets consistently across different sports and modes. Developers must build a small but flexible metadata layer that describes each widget's name, description, required API fields, supported filters, and whether it has Beginner or Advanced versions. This service makes the Widget Library scalable, because new widgets can be added simply by updating metadata instead of touching every part of the system. Without it, the frontend would have to guess or hardcode how widgets behave, which breaks easily. This requirement helps keep the platform structured, expandable, and much easier to maintain long-term.
SFR-23	The system shall adjust the level of detail included in API responses based on whether the requesting user is in Beginner Mode or Advanced Mode.	User mode setting, Widget request Sport type	Simplified dataset for Beginner Mode Detailed dataset for Advanced Mode Mode-labeled API response	Beginner vs Advanced Mode isn't just a display change — it affects how much information comes from the backend, so this requirement creates a clean separation. Developers should build response templates or flags so that widgets request only the level of detail they need. This prevents beginners from being overwhelmed and keeps advanced users

				satisfied with richer detail. It also saves bandwidth by not sending huge payloads unnecessarily. This requirement makes the mode-switching experience feel smooth, intentional, and technically sound.
SFR-24	The system shall implement a unified search index that supports multi-sport searching across players, teams, games, seasons, and widget metadata.	Search text, Sport filters, Player dataset, Team dataset, Game dataset	Ranked search results, Cross-sport result list.	This requirement ties together search, filters, and the Player Library in a single backend feature. Developers should build an index that can handle quick lookups and fuzzy matches for names or abbreviations. Having one search system instead of multiple reduces bugs and ensures consistent results everywhere the user searches. This also opens the door for smarter search features later, like auto-suggestions or filters by role or position. A strong search index gives the platform a professional feel and makes it easier for users to find exactly what they want.

Technical Architecture Note

NashBoard will follow a modular web application structure. A RESTful backend will handle data ingestion, cleaning, and delivery, while the frontend will focus on interactive dashboards and visualizations. The system itself will use caching and API fallback strategies to keep the platform responsive during high-traffic events. Future updates may include machine learning modules to provide predictive analytics, such as matchup outcomes and player performance trends.

UFR ID	Linked SFR	Linked NFR	Notes
UFR-01	SFR-01, SFR-17	SNFR-06, SNFR-16, SNFR-17	Secure account storage and sessions.
UFR-02	SFR-01, SFR-09	SNFR-06	Token-based password reset via email.
UFR-03	SFR-02, SFR-03, SFR-05, SFR-07	SNFR-02, SNFR-03	Sport selection triggers data fetch + cached responses.
UFR-04	SFR-02, SFR-04, SFR-07, SFR-11, SFR-19	SNFR-02, SNFR-04	Live polling, caching, and API error handling.
UFR-05	SFR-01, SFR-05, SFR-14	UNFR-03	Layout saved as JSON per user and reapplied.
UFR-06	SFR-03, SFR-05	SNFR-02	Backend search index and filters.
UFR-07	SFR-03, SFR-05, SFR-07	SNFR-02	Aggregated queries for comparisons and charting.
UFR-08	SFR-01	SNFR-06	Favorites are stored in the user profile.

UFR-09	SFR-05	UNFR-05	Frontend theme state + persistence.
UFR-10	SFR-18	SNFR-18	Export endpoints with streaming file responses.
UFR-11	SFR-05, SFR-07	SNFR-02	Filter parameters applied to DB + cache.
UFR-12	SFR-02, SFR-05	SNFR-02	Different payload/detail levels per mode.
UFR-13	SFR-02, SFR-07	SNFR-02	Aggregation queries for leaderboards/trending.
UFR-14	SFR-01, SFR-05	UNFR-08	Glossary table + tooltip triggers.
UFR-15	SFR-02, SFR-12, SFR-15	SNFR-02	Historical series fetched and rendered in trend widgets.
UFR-16	SFR-15, SFR-18	SNFR-13	Scheduled notification jobs via cron/worker.
UFR-17	SFR-17, SFR-18	SNFR-15	Signed URLs for shared dashboards/widgets.
UFR-18	SFR-12, SFR-08	SNFR-07	Versioned historical data from backups/versions.

UFR-19	SFR-01, SFR-05	UNFR-19	Mode preference is saved and used in rendering.
UFR-20	SFR-05, SFR-12	UNFR-08	Pull explanations from glossary/metadata.
UFR-21	SFR-02, SFR-21	SNFR-02	Rankings computed by the derived metrics engine.
UFR-22	SFR-01, SFR-05, SFR-07	UNFR-02, UNFR-03	Widget metadata + drag/add behavior stored per user.
UFR-23	SFR-02, SFR-03, SFR-05, SFR-12	UNFR-02	Player Library uses normalized data and filters.
UFR-24	SFR-02, SFR-03, SFR-21, SFR-20	SNFR-02	Predictive widgets built on a derived metrics engine, ML-ready.

User non-functional requirements

These non-functional requirements set the standard for how NashBoard should feel as well as perform from the user's point of view. They are the expectations for the speed, accessibility, consistency, and overall user experience of the system.

ID	Requirement (The system shall…)	Measure / Notes
UNFR-	load the main app view in under 3 seconds on a typical connection.	A sports dashboard only feels useful if users can

01		find what they're trying to see without thinking too hard about where it's located. The system should guide users naturally, almost like the layout is "teaching" them as they go. Nothing should feel buried or hidden behind too many clicks, especially for someone who just opened the app for the first time. If switching sports or moving between widgets is confusing, people will assume the entire app is complicated. A smooth navigation flow makes everything else — from searching players to analyzing trends — feel easier and more enjoyable. This requirement basically supports every other feature because users can't appreciate anything they can't easily get to.
UN FR-02	Provide an intuitive UI for non-technical users.	Sports data can be dense, so the interface has to present information in a way that feels calm and clear. Text should be large enough and spaced out enough that users don't

		feel like they're reading tiny numbers jammed together. Charts and widgets need clean colors and simple shapes so important details stand out without overwhelming the viewer. If parts of the interface are too small, too bright, or too cluttered, users will get tired of looking at it quickly. Good readability helps people digest information faster, especially during live games. This requirement makes the whole platform feel more polished and easy on the eyes.
UN FR-03	Be responsive on desktop, tablet, and mobile.	A new user should be able to open the app and understand what's going on within a minute or two. If the system makes people feel lost or unsure, they won't stick around long enough to discover the deeper features. Simple labels, intuitive icons, and a clean first-time layout help beginners feel confident right away. The platform should feel like it "meets the user where they are," rather than

		expecting them to already know how advanced stats work. Over time, users will naturally start exploring more features as they become comfortable. This requirement makes the app welcoming instead of intimidating.
UN FR-04	Keep fonts and visuals clear at all zoom levels.	Users should feel like every page belongs to the same system, with a style that stays consistent whether they're viewing NFL stats or checking the Player Library. Consistent fonts, spacing, colors, and widget layouts make the app feel intentional and trustworthy. When different screens look mismatched, it creates a sense of confusion or lack of polish. Developers should reuse patterns where possible so users don't have to relearn how to interpret each new page. A consistent design also helps users navigate faster because they know what to expect visually. This requirement ties the whole experience together.

UN FR-05	allow seamless switching between light and dark mode.	Even if some operations take time behind the scenes, the interface should feel fast and responsive. Users shouldn't feel like they're waiting unnecessarily, especially when switching sports or opening a widget. Loading indicators, small animations, or cached data can help the system feel smoother. If the app feels slow, users will assume the data is outdated or unreliable. Quick feedback also keeps users more engaged while watching live games. This requirement shapes the "feel" of the app more than the raw speed.
UN FR-06	meet WCAG 2.1 AA accessibility guidelines where feasible.	Users should never feel like the screen is crowded or stuffed with more information than they can handle at once. Even though the platform handles complicated stats, those details need space to breathe so they don't overwhelm people. Good spacing and clear grouping help users understand what relates

		to what. Too much clutter makes everything harder to read and interpret. Simplifying the layout also makes the dashboard feel more professional and less like a spreadsheet. This requirement is key to making advanced analytics feel approachable.
UN FR-07	Support color-blind friendly palettes and avoid relying on color alone.	Beginners should never feel embarrassed or confused by what they're seeing. The system needs to feel friendly and encouraging, offering simple explanations or hints when things look complicated. Beginner Mode exists to help users breathe a little easier and not feel pressured to understand every stat immediately. The platform should help them enjoy the sport more rather than making them work to understand the interface. When beginners feel comfortable, they're more likely to explore deeper features at their own pace. This requirement helps widen the app's appeal far beyond hard-core

		fans.
UN FR-08	Provide tooltips and explanations for advanced stats in plain language.	The interface should work well for users with different vision needs, device sizes, or interaction styles. Simple things like good color contrast, readable fonts, and large tap areas make a big difference. Users shouldn't have to fight the interface just because they're on a smaller device or have a different accessibility setting. Even though this requirement doesn't add new features, it improves the experience for everyone. It also shows that the platform respects all users, not just one specific group. Accessibility helps the platform feel more thoughtful and inclusive.
UN FR-09	Give immediate UI feedback to user actions (clicks, searches, etc.).	Buttons, toggles, and menus should always behave the way users expect them to. If a feature works one way in one place and differently somewhere else, users can get frustrated or lose trust in the system.

		Predictability helps users build mental habits, so they know what each section of the app will likely do. This also reduces the time it takes to learn new parts of the interface. Predictable behavior makes the whole app feel more stable and reliable. This requirement indirectly supports ease of use and overall user satisfaction.
UN FR-10	show human-readable error messages.	Users shouldn't feel mentally drained while trying to explore stats or compare players. The system should break information into digestible chunks rather than overwhelming users with everything at once. Clean grouping, clear labels, and simple visual cues help users understand complex information more easily. Reducing cognitive load makes the platform feel enjoyable instead of like a homework assignment. Over time, this encourages users to come back more often because it feels effortless to use. This requirement connects to the platform's core goal

		of making analytics accessible.
UN FR-11	enforce a session timeout of at least 30 minutes of inactivity.	A session timeout might not be something users ever think about, but it matters behind the scenes for keeping accounts safe. The idea is to strike a balance between convenience and protection: users shouldn't get kicked out too quickly, especially if they're watching a game and checking stats on and off. At the same time, leaving a session open forever on a shared device isn't a great idea either. A 30-minute limit feels fair because it gives people plenty of room to browse casually without forcing constant logins. This requirement creates a little safety net that quietly works in the background without interrupting the experience. It helps protect users without making them feel restricted.
UN FR-12	Maintain UI consistency across all pages and sports.	The platform should feel like one unified system, not a collection of mismatched pages pulled from different

		apps. Users shouldn't feel like they're "learning a new layout" every time they switch sports or open a different section. Keeping fonts, spacing, buttons, and menus consistent gives the whole app a calmer, more polished feel. When the UI stays steady across features, people develop trust in the platform because nothing feels out of place. Even small details — like where filters appear or how charts are labeled — help users feel grounded. This requirement keeps the entire interface predictable and easy to navigate.
UN FR-13	support a minimum of 50 concurrent users without noticeable slowdowns.	Even if the system is still growing, it should be able to handle a classroom's worth of users or a group of friends checking stats at the same time. Nobody wants to wait for pages to load just because a few other people logged in. Hitting this baseline helps ensure the platform doesn't stumble during normal use, even when games

		are happening. If the app slows down every time traffic bumps up a bit, users will assume the whole thing is unstable. Meeting this requirement builds confidence that the platform won't crumble under everyday pressure.
UN FR-14	Provide optional tutorial overlays for new users.	A lot of people who open NashBoard aren't experts, and jumping straight into advanced stats can feel intimidating. A simple, optional tutorial overlay can ease that tension by pointing out where things are and what they do, almost like a quick tour. It shouldn't be long or overwhelming — just a light guide that gives beginners enough direction to feel comfortable. Users should be able to skip it if they want, but those who do try it should come away feeling more confident. This requirement helps the platform start off on the right foot with new users and prevents confusion early on.

UN FR-15	Keep navigation depth shallow (no more than 3 levels deep for major actions).	The fewer clicks it takes to reach something, the easier the whole app feels. Sports fans don't want to dig through endless menus when all they want is one matchup or stat. Keeping navigation shallow avoids that buried feeling and allows users to jump around without losing track of where they are. When everything is reachable within a few taps or clicks, the platform feels lighter and more user-friendly. This requirement keeps the experience clean and prevents frustration from overly complicated menu structures.
UN FR-16	ensure the app remains usable on lower-resolution screens.	Not every user has a brand-new device or a big monitor, so the platform needs to adjust gracefully to smaller or older screens. Text should stay readable, buttons should still be easy to tap, and charts shouldn't look cramped or broken. The goal is to make sure the app still

		feels good to use even if the screen isn't ideal. If certain widgets become unreadable on smaller displays, users might assume the platform wasn't made for them. This requirement makes the dashboard feel inclusive and reliable for all kinds of devices.
UN FR-17	Keep major interactions (widget add, filter apply) under 1 second when cached.	Actions like adding a widget or applying a filter should feel almost instantaneous, especially when the data is already cached. These small interactions happen constantly as users explore the dashboard, so any delay becomes noticeable very quickly. Quick responses make the interface feel alive and responsive, which is important during live games when things move fast. If these common actions are slow, it can make the whole app feel sluggish. This requirement keeps the experience snappy and enjoyable.
UN FR-18	Provide clear indications when data is historical vs live.	Users shouldn't have to guess whether they're looking at real-time stats or an older snapshot. A

		simple label or small visual cue can prevent confusion and help people understand what they're actually seeing. This is especially important when comparing past seasons or looking at player trends, because mixing live stats with old ones can skew interpretation. Clear indicators keep the information honest and make the platform feel more transparent. This requirement helps users trust the data they're looking at.
UN FR-19	Make Beginner Mode clearly visible and easy to toggle.	Beginner Mode is one of the platform's biggest advantages, so users should never struggle to find it. The toggle needs to be easy to access and clearly labeled so newer fans feel supported. If the mode is hidden or confusing, people might not even realize the feature exists, which defeats its purpose. A simple, friendly switch makes the platform feel inviting and lowers the pressure on users who aren't comfortable with advanced analytics. This requirement helps bridge the gap between

		casual fans and stat-savvy users.
UN FR-20	Avoid jargon in user-visible text, or explain it immediately.	Sports analytics are full of phrases that can confuse newcomers, and the platform shouldn't assume everyone knows them. If a term must be used, it should be explained right away or linked to the glossary. This keeps the tone approachable and prevents the app from feeling like an advanced math class. When language stays clear, users feel more confident and willing to explore the deeper parts of the dashboard. This requirement supports the core mission of making analytics accessible to everyone.

System non-functional requirements

This section covers the performance, reliability, and security expectations on the system side. These requirements make sure NashBoard can handle traffic smoothly, keep data secure, and remain stable as the platform grows.

ID	Requirement (The system shall…)	Category
SNF R-01	Respond to standard API requests in under 500 ms under normal load.	Performance

	Performance shapes how “alive” the platform feels. Even if the backend is doing a lot of work, the app shouldn’t make users feel like they’re waiting around. When actions take too long, it breaks the sense of flow and makes the experience feel outdated or frustrating. Developers can lean on caching and preloading to make common tasks feel much faster from the user’s point of view. This requirement isn’t about making everything perfect — it’s about keeping the app feeling smooth and modern. A system that responds quickly naturally feels more trustworthy and enjoyable to use.	
SNF R-02	Maintain an average API response time under 1 second at peak load. Sports APIs go down more often than users might expect, especially during big games when everyone is checking stats. The platform shouldn’t crash or show empty screens when that happens. Instead, it should fall back to cached data or a friendly message explaining the situation. Users appreciate honesty and stability even when the data isn’t fully up to date. This requirement keeps the app feeling steady during unpredictable moments. It shows that the platform was built thoughtfully, with real-world conditions in mind.	Reliability
SNF R-03	achieve a cache hit rate of at least 70% for common queries. Security is one of those things users don’t notice until something goes wrong, so it needs to be handled quietly and carefully. Even though a sports dashboard isn’t as sensitive as a banking app, people still expect their accounts to be protected. Using hashing, secure cookies, and encrypted traffic helps prevent basic security risks without making the login experience harder. Good security also builds trust because it shows respect for the user’s information. This requirement keeps the platform safe while staying out of the user’s way. It supports every part of the account	Security

	and preference system.	
SNF R-04	handle traffic spikes during major games without a complete outage. Traffic spikes during primetime games are unpredictable but guaranteed, so the platform needs to handle them smoothly. Users shouldn't feel like the app "breaks" when everyone logs in at once. Developers can prepare for this by designing services that distribute load or temporarily limit heavy operations. A scalable design also makes it easier to grow the platform over time without rebuilding everything. This requirement keeps the app stable during the moments when users rely on it most. It gives the system room to grow without hitting constant performance walls.	Scalability
SNF R-05	support horizontal scaling of backend services. A maintainable system saves huge amounts of time in the long run. When everything is tangled together, even small changes can create unintentional bugs elsewhere. A modular structure allows developers to work on isolated areas without causing ripple effects. It also makes onboarding new team members much easier because they can understand parts of the system without needing full knowledge of everything. This requirement helps keep the codebase healthy and organized as the platform grows. A modular design is one of the most important qualities of scalable software.	Maintainability
SNF R-06	Encrypt sensitive data at rest and in transit. Sports data changes constantly, and users want to know whether what they're seeing is live or slightly delayed. Without clear timestamps, people might misinterpret stats or assume the app is wrong. Making the update cycles transparent helps users trust the	Data Quality

	information even if external APIs slow down. Developers should avoid silently mixing data from different times, which can lead to confusion. This requirement ensures that the platform always presents accurate and honest information. It reinforces the platform's reliability and credibility.	
SNF R-07	perform nightly backups with at least 7 days of retention. Logs help developers understand what's happening inside the system at any moment. They're especially valuable when something unexpected occurs or when an error appears only for certain users. Without proper logging, debugging becomes a guessing game. Good monitoring helps spot problems early so they don't grow into larger issues. This requirement makes the system easier to maintain and keeps developers prepared for anything. It quietly supports the long-term health of the platform.	System Monitoring
SNF R-08	support data recovery within agreed RPO/RTO goals (e.g., RPO 24 hrs, RTO 4 hrs). Users access sports dashboards from all sorts of devices — some new, some old — and the app needs to adapt gracefully. The interface shouldn't break just because someone uses a smaller screen or an older laptop. Ensuring compatibility makes the platform feel welcoming instead of exclusive. Developers should test the system on a few different devices to catch issues early. This requirement helps keep the experience consistent for everyone. It shows that the platform was designed for real users, not just ideal scenarios.	Compatibility
SNF R-09	log key system events (errors, API failures, auth events) centrally. Frontend components depend heavily on consistent	API Stability

	backend responses. If the backend starts returning unpredictable shapes or missing fields, the UI can break or show confusing errors. A stable API helps the entire platform feel smoother because every component knows what to expect. Even during failures, returning a predictable fallback structure keeps things under control. This requirement makes integration between frontend and backend much more reliable. It keeps the platform stable even when things aren't perfect behind the scenes.	
SNF R-10	Provide metrics for CPU, memory, and error rates. Not every feature will work perfectly all the time — that's just how real systems behave. When something temporarily fails, the app should still offer a useful experience instead of completely shutting down. This might mean showing cached data or a simplified version of a widget. Users are generally forgiving as long as the platform remains usable and communicates clearly. Graceful degradation keeps the dashboard steady during bumps in the road. It shows users that the system was designed thoughtfully and with resilience in mind.	Graceful Degradation
SNF R-11	Support CI/CD for deployment and testing. The system should let developers or admins adjust settings without having to rewrite major parts of the code. Small things like refresh intervals, caching rules, or API keys shouldn't require digging deep into the backend just to make changes. When configuration is flexible, updates feel lighter and safer, and the system becomes easier to tune over time. This also helps during maintenance because small adjustments can fix problems without big deployments. Even though users never see this directly, it improves stability in the long run. It makes the platform more adaptable to future features or changes in external APIs.	Configuration

SNF R-12	Maintain versioned public API documentation. Users should feel confident that the stats they're looking at are reasonably up to date, especially during live games. The system doesn't have to be perfect, but it should follow predictable update patterns so users aren't left guessing whether the information is current. Clear timing rules help keep things consistent across widgets and sports. If updates fall behind for any reason, the app should make that clear instead of pretending everything is fine. This requirement protects the user's trust in the platform's accuracy. It's also important for comparisons and trends since outdated data can lead to misunderstandings.	Data Quality
SNF R-13	Raise automated alerts for failures and anomalies. APIs fail unpredictably, so the system needs a steady way to bounce back when they do. Users shouldn't have to refresh or reopen the app just to get data flowing again. The backend should try reconnecting automatically and smoothly resume normal updates once the API stabilizes. This quiet recovery avoids disrupting the user's experience. It's also helpful during live events when API traffic gets intense. This requirement helps the system feel resilient and reduces the sense of "things breaking" during peak moments.	Recovery
SNF R-14	Implement exponential backoff for failed external API calls. Errors shouldn't spill out into the user interface in a way that scares or confuses people. Instead, the system should handle problems behind the scenes and only show simple, calm messages when something truly needs attention. Developers should make sure errors are logged properly so they can be investigated later. Good error handling prevents small issues from becoming bigger ones. It also helps users feel like the platform is stable and dependable. This requirement	Fault Tolerance

	keeps the experience smooth even when something unexpected happens.	
SNF R-15	avoid exposing stack traces or sensitive details to clients. Even when conditions change, internal APIs should always return predictable structures so the frontend doesn't have to guess what kind of data it's receiving. When responses stay consistent, it makes the UI code cleaner and easier to maintain. Inconsistent responses create moments where widgets might break or misinterpret values. Sticking to a reliable structure reduces bugs and weird edge cases. This requirement also makes future updates easier because backend changes won't disrupt existing frontend logic. Overall, it keeps communication between the two layers steady and stress-free.	API Stability
SNF R-16	Sanitize all user input to prevent injection and XSS. The system should quietly keep an eye on itself by watching for unusual patterns like API delays, failed jobs, or unusual spikes in error rates. When something crosses a threshold, the system should alert the developers so they can respond quickly. This prevents issues from lingering unnoticed until users start complaining. It also helps catch small problems before they grow into outages. Monitoring creates a safety net that improves reliability without requiring constant manual checking. This requirement supports the long-term health and stability of the platform.	Monitoring
SNF R-17	Enforce role-based access control (RBAC) for admin operations. The system shouldn't overuse memory, CPU, or bandwidth, especially during high-traffic periods. Efficient design helps keep the platform fast even when	Resource Efficiency

	demand increases. It also lowers the risk of crashes or sudden slowdowns during popular sporting events. Developers should build services that scale smoothly and avoid unnecessary work behind the scenes. Good resource management also reduces hosting costs and improves long-term sustainability. This requirement quietly improves both performance and stability without users ever noticing directly.	
SNF R-18	auto-rotate credentials (API keys, secrets) regularly (e.g., every 30 days). Widgets should always load in a predictable order and layout so users don't feel like the interface is jumping around. If some widgets take longer to load than others, the app should handle that gracefully instead of letting things shift unexpectedly. This helps keep the dashboard feeling stable and intentional. Users should trust that the platform won't rearrange their layout randomly or break the flow of information. This requirement strengthens the overall feeling of polish and smoothness. It also helps the interface feel more reliable during live games.	UI Stability
SNF R-19	Provide a centralized metrics dashboard for system health. Even though users never see it, good internal documentation makes the entire system easier to maintain and expand. Future developers shouldn't have to guess how a service works or why certain decisions were made. Clear explanations help reduce mistakes and speed up feature development. Documentation also helps keep the platform stable as more people contribute over time. This requirement ensures the system stays understandable rather than becoming a confusing mix of old and new code. It supports the long-term success of the project.	Developer Support

SNF R-20	Use load balancing to distribute incoming traffic. Data coming from different sources should look and behave the same way once it reaches the frontend. Without consistent formatting — like uniform date types, team names, or stat abbreviations — widgets could show mismatched or confusing information. Users shouldn't have to wonder why two sections display the same stat differently. Consistency also makes debugging easier because developers don't have to track down formatting differences. This requirement helps keep the entire interface clear and trustworthy. It keeps the system's data presentation clean and predictable.	Data Formatting

Optional- User domain requirements

The user domain requirements explain how the advanced statistics will be presented in a way that's approachable for casual fans. These focus on context, explanation, and storytelling to help users actually understand what the numbers mean.

ID	Requirement	Notes
UDR-01	Stats shall be explained in plain language with access to a glossary.	Sports fans check dashboards during high-energy moments, so they expect updates to feel almost immediate. Even if the system can't be perfect, it should deliver data frequently enough that users don't feel behind the game. This expectation comes from common behaviors — people flipping between TV, social media, and stats at the same time. If the dashboard feels too slow compared to what they see on TV, they'll assume something is wrong. This requirement reflects real user habits, not just technical design. It sets the baseline for how responsive the platform needs to feel.

UDR-02	Advanced stats shall be broken down into simpler components when possible.	Fans rely on familiar terms, and if the app uses strange or inconsistent labels, it can create confusion or frustration. People are used to the way ESPN, CBS, NFL Network, and NBA broadcasts describe stats, so sticking to those conventions helps the platform feel more natural. This requirement makes the system easier to learn because users recognize the language right away. It also adds credibility, as mismatched labels can make a system look amateurish. This expectation comes directly from how real sports media presents information.
UDR-03	Users shall be able to switch between basic and advanced views.	Fans compare eras, players, and seasons, so they expect old data to remain exactly as it was originally recorded. Changing historical numbers or mixing updated API data with old seasons can lead to confusion or arguments. Keeping past data “locked” helps users trust long-term analyses. Sports fans notice small stat inconsistencies more than most audiences. This requirement reflects the domain’s obsession with accuracy and tradition.
UDR-04	Key insights (e.g., “great matchup”, “cold streak”) shall be summarized in short sentences.	Sports fans frequently switch between phones, laptops, and tablets while following games. They expect their dashboards, favorites, and mode settings to follow them automatically. If personalizations disappear depending on the device, it feels like the platform is unreliable or incomplete. This requirement matches how modern sports apps behave. It reflects actual user behavior patterns — people watch games everywhere.
UDR-05	Historical context (e.g., last season vs this season) shall be available for key views.	Fans often follow more than one sport, so requiring separate logins would feel annoying and outdated. They should be able to switch from NFL to NBA to MLB within the same session easily. This reflects the

		reality of multi-sport fandom and how platforms like ESPN, Sleeper, or CBS Sports function. If switching sports feels like switching apps, users will lose interest. This requirement ensures the experience stays unified.
UDR-06	Educational overlays or hints shall be available to guide users through complex views.	Many fans are curious about analytics but don't fully understand them. Instead of forcing users to guess, the platform should help guide them with simple explanations. This reflects a real trend: sports broadcasts are using analytics more, but they often pause to explain them. Providing this clarity makes the platform feel smarter but still approachable. This requirement comes from the domain's shift toward data literacy.
UDR-07	The system shall highlight important streaks and records automatically.	Injuries affect fantasy decisions, bets, and general understanding of a team's situation, so this information must be presented clearly. Users expect injuries to stand out instead of blending in with normal stat updates. This reflects real-life behavior — people check injury reports constantly on game days. If injuries are buried or unclear, the platform will feel unreliable. This requirement is driven by the domain's heavy emphasis on player availability.
UDR-08	Injury updates shall be filterable and clearly labeled.	Fans rarely type full player names perfectly. They often use abbreviations or casual nicknames like "JT," "Luka," or "CMC." This behavior is specific to sports communities where shorthand is extremely common. If the search doesn't recognize these patterns, users will assume the system is rigid or poorly designed. This requirement comes straight from how fans talk about sports every day. It ensures the system aligns with real-world user habits.
UDR-09	All data shall be tied to verified sources	When fans compare players or teams, they assume the system is using the same definitions and time

	(trusted APIs).	ranges across both sides. If one widget pulls season averages and another uses last-10-games by accident, the comparison won't feel fair or trustworthy. Fans notice inconsistencies quickly, especially when comparing their favorite players. This expectation comes from how TV broadcasts and major sports sites always use consistent baselines during analysis. Keeping comparisons aligned helps users make sense of the data instead of questioning it. This requirement supports both fairness and clarity in the analytics.
UDR-10	Storytelling features (e.g., "biggest play of the game") shall be supported.	During live games, fans care more about what is happening *right now* than anything else. They expect the dashboard to highlight big plays, scoring changes, turnovers, injuries, and other momentum-changing events. If the system hides or delays these updates, it will feel out of sync with the actual game. This expectation reflects how fans watch sports — checking apps during critical moments to stay connected. Making key events obvious helps users feel like the platform is alive and reacting with the game. This requirement keeps the experience immersive and aligned with real fan behavior.

Optional- system domain requirements

This list of system domain requirements describes how NashBoard's backend will handle complex sports data from different providers. The focus here is on structure, flexibility, and data integrity across multiple sports and sources.

ID	Requirement	Notes
SDR -01	Support multiple data providers	The sports data domain is messy — every provider formats its stats differently. Some give clean JSON,

	with configurable fallback order.	others mix nested fields or use unusual naming conventions. The system must blend these differences into a unified structure before widgets can use them consistently. This requirement comes not from users but from the domain's fractured data ecosystem. Without normalization, the platform would feel chaotic and inconsistent. It's a technical reality of sports analytics.
SDR -02	Normalize schemas across sports where logically possible.	API providers usually restrict how often the system can request updates, especially for free or academic tiers. This limitation is part of the domain itself and can't be ignored. Developers need to design around these rules without hurting user experience. High-pressure moments — like the last 5 minutes of a close game — often strain API request limits. This requirement ensures the system respects domain constraints while still feeling "live" enough for users.
SDR -03	Use unique global IDs for teams, players, and games.	Advanced analytics like True Shooting Percentage or EPA (Expected Points Added) don't come ready-made from most providers. The system must compute them using formulas built from multiple API fields. This requirement comes from the domain's nature — analytics often require calculated stats. Without this processing, Advanced Mode would feel incomplete or misleading. It ensures the platform reflects actual sports analysis conventions.
SDR -04	Schedule ingestion jobs separately per sport.	Sports stats reset each season, and combining data across seasons without clear labeling can confuse users. The domain treats seasons as separate analytical units. The system must respect this rule to keep comparisons meaningful. Mixing seasons could make players look better or worse than they actually are. This requirement comes from long-standing sports statistical traditions.

SDR -05	Validate incoming data automatically and reject malformed records.	Unlike scores or play-by-play data, injury reports are sometimes slower and rely on manual updates. This is a domain limitation outside the system's control. The platform needs to tell users when injury data might lag behind other updates. This requirement ensures transparency and accuracy even when the underlying data isn't perfect. It aligns with real-world sports reporting practices.
SDR -06	Record data lineage (source, timestamp, transformation).	Sports rosters change constantly due to trades, waivers, or injuries. When a player moves, the system must update references across multiple historical or current datasets to avoid mismatches. This is a domain reality — players don't stay in one place. If data is not updated properly, users may see players listed on the wrong team. This requirement keeps the platform aligned with the fast-moving nature of sports.
SDR -07	Support analytical (OLAP-style) queries for trends.	Sports leagues operate across multiple time zones, and schedules often reference official times instead of local user time. The system must interpret and convert times accurately. This requirement comes directly from the sports scheduling domain, where mistakes lead to confusion about game starts. Users rely on accurate timings to plan their viewing. Domain rules around game times are strict and must be respected.
SDR -08	Store immutable raw data snapshots before transformation.	A "rebound" in basketball has nothing to do with a "rebound" in hockey, and "yards" in football don't exist in soccer. The system must respect the statistical vocabulary of each sport. This requirement comes from the diversity of sports analytics — each sport has unique structures. If the system accidentally mixes definitions, users will spot errors instantly. This ensures data stays meaningful in its proper context.

SDR -09	Run quality checks after each ingestion run.	Different sports break down events in completely different ways — an NFL “drive” has nothing in common with an NBA “possession,” and baseball events follow their own rhythm entirely. The system must understand the event structure of each sport to avoid misinterpreting plays or mislabeling actions. This requirement exists because providers don’t standardize these structures across leagues. If the system mixes meanings, widgets could show misleading or nonsensical information. Handling event structures correctly ensures the analytics stay authentic to the sport.
SDR -10	Provide a plug-and-play architecture to add new sports with minimal changes.	Many leagues issue stat corrections hours after games end — for example, adjusting assists, tackles, or turnovers based on post-game review. This means the system must be able to update previously recorded data without breaking history or confusing users. These corrections are a built-in part of the sports domain, not a system error. Developers must design storage and update processes that allow safe retroactive changes. This requirement helps the platform stay aligned with official league data and maintain its credibility over time. It prevents cases where users think the platform’s numbers are wrong when corrections were simply pending.

UFR ID	Linked SFRs	Linked UNFRs	Linked SNFRs	Linked UDRs	Linked SDRs
UFR-01 Create an account	SFR-01, SFR-17, SFR-09	UNFR-01, UNFR-11	SNFR-03, SNFR-06,	UDR-01	SDR-02

			SNFR-11		
UFR-02 Reset password	SFR-01, SFR-09, SFR-17	UNFR-01	SNFR-03, SNFR-06	UDR-01	SDR-10
UFR-03 Select a sport & load dashboard	SFR-02, SFR-03, SFR-05, SFR-07	UNFR-03, UNFR-05	SNFR-01, SNFR-02	UDR-02, UDR-05	SDR-01, SDR-03
UFR-04 Display live scores/stats/injuries	SFR-02, SFR-04, SFR-07, SFR-11, SFR-19	UNFR-05, UNFR-18	SNFR-01, SNFR-02, SNFR-06	UDR-01, UDR-07, UDR-10	SDR-02, SDR-05
UFR-05 Customize dashboard layout	SFR-01, SFR-05, SFR-14, SFR-22	UNFR-02, UNFR-03	SNFR-05, SNFR-18	UDR-04	SDR-01
UFR-06 Search players/teams/games	SFR-03, SFR-05, SFR-24	UNFR-09	SNFR-09	UDR-08	SDR-01, SDR-09
UFR-07 Player/team comparisons	SFR-03, SFR-05, SFR-07, SFR-21	UNFR-10, UNFR-18	SNFR-06	UDR-02, UDR-09	SDR-03, SDR-04
UFR-08 Favorite teams/players	SFR-01	UNFR-03	SNFR-03	UDR-04	SDR-01

UFR-09 Toggle light/dark mode	SFR-05	UNFR-02, UNFR-04	SNFR-01	UDR-04	—
UFR-10 Export PDF/CSV	SFR-18	UNFR-17	SNFR-05	UDR-03	SDR-04
UFR-11 Filtering system	SFR-05, SFR-07	UNFR-02, UNFR-05	SNFR-01	UDR-02	SDR-01
UFR-12 Beginner/Advanced Mode	SFR-23	UNFR-03, UNFR-19	SNFR-05, SNFR-11	UDR-06	SDR-03
UFR-13 Leaderboards & trending	SFR-02, SFR-07, SFR-21	UNFR-02, UNFR-17	SNFR-01	UDR-02, UDR-10	SDR-03
UFR-14 Glossary definitions	SFR-01, SFR-05	UNFR-08, UNFR-20	SNFR-12	UDR-06	—
UFR-15 Trend charts	SFR-02, SFR-12, SFR-15	UNFR-02	SNFR-06	UDR-03	SDR-04
UFR-16 Notifications	SFR-15, SFR-18	UNFR-14	SNFR-07, SNFR-13	UDR-07	SDR-05
UFR-17 Share dashboards/widgets	SFR-17, SFR-18	UNFR-09	SNFR-03, SNFR-	UDR-04	SDR-01

			09		
UFR-18 View past stats & history	SFR-12, SFR-08	UNFR -18	SNFR-06, SNFR-07	UDR-03	SDR-04, SDR-10

Detailed Rationale of the user functional, non-functional. Optional domain requirements

The way I structured the requirements came from thinking about how real fans actually use sports data. A lot of people want more information, but not everyone has the time or background to dig through endless stat sheets. So the user requirements are focused on simple tools that make sense — things like filters, favorite lists, and quick explanations right where they're needed. On the system side, it's all about keeping the app steady and fast, because if it lags during a game, nobody's going to stick around. The non-functional requirements are there to make sure it works smoothly on any device, whether someone's at home or checking on their phone at a bar. And the domain pieces are what tie it all together — giving the numbers meaning, not just data points. I want fans to feel like they're learning something, not decoding a math problem.

Detailed application of the code of ethics to the specification phase

1. Public - Every choice made by an engineer will be made with the users and the community in mind. An engineer working on this project will develop the platform to protect user privacy, avoid deceptive design, and give people clear, honest information.
2. Client & Employer - All of the work that is done on this project will support its main goals and the goals of the people funding it, in respect to legal requirements, user protection, and ethical guidelines. The interest of business will, in no circumstance, outweigh user trust or data security.
3. Product - Code, its infrastructure, and its system updates will be met with professional standards. Every feature of the program will be tested rigorously and verified before release to make sure the product is stable.
4. Judgment - An engineer who works on this project will rely on their own honesty, honor, and professional judgment when reviewing features or implementing them. No feature of this program shall be approved if it is a risk to the user's security or ethical standards that we uphold

5. Management - Engineers who work on this project will keep an open and transparent process throughout the development period. The project will focus not just on speed but rather on accessibility, communication, documentation, and performance as well.
6. Profession - Any engineer who works on this project will be aiming to build a project that reflects our determination to commit a valuable product to the software engineering field. By implementing clean, maintainable code, we can provide a good example of solid engineering to the community, something we want to reflect.
7. Colleagues - The culture surrounding this project will be built and rely on the use of fairness and respect, especially when encountering feedback of any kind.
8. Self - Engineers who work on this project are expected to keep learning and growing in whatever they encounter throughout their time working on this project. They are to stay current with the news, the most efficient security practices, and data ethics, as they will change throughout time.

Details about your client, meetings, outcome of the meetings

A lot of what NashBoard is came straight from real, tough conversations with close friends and other casual sports fans. I didn't hand them a survey, and rather than ask them questions like what do you want to see in a software, I started asking them what they didn't like about the current websites they use. After that, it wasn't complicated, and they clearly conveyed to me that they wanted something easy to read, clean to look at, and quick to load. Nobody said they needed wild features or some super complex dashboard. So that is what I built around, and anything that comes afterwards will be at the will of my friends (clients). All in all, I paid attention to what they found themselves complaining about. “I hate when the site lags”, or “ I just want to see this one matchup without digging,” were commonplace when asking my friends about this project, and it was this feedback that helped shape the idea of NashBoard into what it is currently. It’s not a guess at what users might like. It’s what they actually told me.