Project direction over time:

Please list out changes in the directions of your project if the final project is different from your original proposal (based on your stage 1 proposal submission).

The direction of our project shifted from a fully concert-focused, multi-city travel planner to a more localized, event-centered application due to constraints in the data we used. Originally, we planned to calculate total costs including ticket prices, allow users to search by artist name, filter by state, and offer genre-based recommendations through the "Surprise Me" button. In the final project, we removed ticket pricing because real-time prices were unavailable and ticket prices vary largely within each event based on seating. We also limited searches to event names only. We also focused exclusively on Chicago Airbnbs since other cities were not included in the dataset, and because the dataset was already long enough. The "Surprise Me" feature was simplified to randomly select an event instead of a genre because Ticketmaster does not include the genre in their dataset and because we expanded to general events rather than music concerts. Our map visualization was also adjusted to show only the top search result event venue and the top five nearby Airbnb listings. These changes made the application more realistic and aligned with the actual data we could obtain while still supporting cost-based planning for events.

Project Reflection (What we achieved/failed for usefulness)

Discuss what you think your application achieved or failed to achieve regarding its usefulness.

<u>What We Successfully Achieved</u>

1. Solved a Real Problem
    a. We successfully addressed the core pain point: comparing concert trip costs across multiple cities. Before GetLitty, a fan would need to:
        - Check Ticketmaster for events in City A, B, C...
        - Open Airbnb separately for each city
        - Use Google Maps or another external service to see proximity of Airbnb to venue
        - Manually calculate total costs
        - Keep track in a spreadsheet
    b. GetLitty centralizes this into one search with automatic cost calculations, saving users significant time and effort.
2. Actionable Data Integration
    a. We successfully merged two datasets (Ticketmaster events and Airbnb listings) in a way that produces useful insights:
        - The distance filter ensures lodging is actually convenient to the venue
        - We give users listing IDs as well so they easily find and book the Airbnbs
        - Giving users a way to keeping track of what events they want to go to and track if they've figured out accommodations

    b. This is more useful than either dataset alone.
3. Practical Features for Real Users
    a. Search & Filter System:
        - Filters by event, date range, and distance create a flexible search that matches how people actually plan trips
        - The distance slider (1-200 miles) lets users decide their own convenience vs. cost tradeoff
    b. "Surprise Me" Feature:
        - Addresses the common problem: "I want to go to an event or concert but don't know who's touring" in a way that makes the user feel entertained and have fun.
        - Provides discovery with affordability as the guide
    c. My Events Management:
        - The trigger preventing double-bookings has direct real-world volume so it prevents users from making impossible plans
        - Viewing and managing saved events helps users track their concert calendar
        - Bulk-add city feature is useful for festival-goers who want to attend multiple days of a multi-day event
        - Allows the user to record if they've already booked accommodations for the event(s) that they want to attend; helps people ensure they have proper planning even if they book events far in advance
4. Cost-Conscious Focus
    a. Every query prioritizes affordability, which aligns perfectly with our target audience: music fans who want to travel to concerts but have budget constraints. The "cheapest total cost" metric (ticket + lodging) is the most important number for decision-making, and we surface it prominently. Ties in the cheapest cost are broken by distances.

<u>What We Failed to Achieve or Could Improve</u>

1. Limited Geographic Coverage
    a. For the purpose of this project, we decided to focus on Airbnb data, and due to the extremely large size of the Chicago dataset, we decided to just focus on Chicago too. We didn't see the point in cleaning the data for so many more cities because it would be so much data, and hence make the application a lot slower.
    b. Impact on Usefulness:
        - A user searching for a concert in Wyoming or Montana gets incomplete results.
        - Recommendations might suggest expensive cities simply because we lack data for cheaper nearby alternatives.

    c. What We Could Have Done:
        - Scraped additional lodging sources (hotels.com API, Booking.com)
        - Adding more data for other cities can be a future improvement
2. Static Data causes Outdated Recommendations
    a. Our data is a snapshot (Airbnb prices from when we loaded the dataset, events from when we queried Ticketmaster)
    b. Impact on Usefulness:
        - Airbnb prices fluctuate based on demand - a concert announcement can cause prices to spike
        - Events sell out, get cancelled, or add dates
        - Our "available" listings might not actually be available anymore
        - Price recommendations could be dangerously misleading if prices have increased 2x since our data load
    c. What We Could Have Done:
        - Implemented live API calls to Airbnb/Ticketmaster instead of static database imports
        - Added "last updated" timestamps so users know data freshness
3. Map Feature Underutilized
    a. The map only shows top 5 Airbnb listings for a selected event
    b. What It Should Do:
        - Show all events on one map with cost-based color coding (our original creative component vision)
        - Let users click regions to filter events
        - Show both venue AND lodging clusters
        - Display travel distance/time between lodging and venue
    c. Impact on Usefulness: Users still rely on the table, making the map feel underutilized.
4. No Social/Review Integration
    a. Cost isn't everything
    b. Missing Context:
        - No Airbnb reviews/ratings (is this cheap place a dump?)
        - No venue reviews (is this venue terrible?)
        - No artist popularity data (is this worth traveling for?)
    c. Impact on Usefulness: Users might book the "cheapest" option only to have a terrible experience, damaging trust in our recommendations.

Each team member should describe one technical challenge that the team encountered. This should be sufficiently detailed such that another future team could use this as helpful advice if they were to start a similar project or where to maintain your project.

1. Uploading the databases onto GCP (and learning how to navigate GCP in general) - Ashley
    a. We had difficulty with getting the data from ticketmaster API and airbnb csv into the tables, because columns would match up, but it still wouldn't import
    b. We had to go to office hours and then we understood the foreign key constraints were causing the issues, so we recreated the tables without foreign key definitions, and added them back in after populating the tables
    c. We recommend that if someone is having similar issues where they are unable to populate the tables in the database even when the columns seem to match up exactly and they have verified that, then they should make sure to check the table information to see if there may be foreign keys that could be getting in the way. If there are, then we recommend starting over and deleting that table, and recreating it without the foreign key definitions. Then after populating the table, they should add the foreign keys back.
2. Creating the initial database connection - Michelle
    a. We had issues with connecting our database on GCP to the actual code for the website because we weren't properly authorizing our IP address.
    b. We went to office hours and realized it was because we had to whitelist the IP address in GCP. We also learned that we needed to turn on the GCP in the first place for the connection to work. While it may seem straightforward, for people who are not used to GCP, they may not realize that they need to do that. But ultimately in office hours we learned that whitelisting was also an issue, so it was a good thing that we went.
    c. We also realized that we needed to be mindful to turn off the GCP when we finished to ensure we had enough credits. This was something we had trouble with later in the semester, so we had to redeem more credits with help from a TA. We recommend that future teams fully understand how the GCP works before trying to connect the backend to the database.
3. Normalizing the data - Ana
    a. Having to go back and forth and finding exact names for columns was a challenge, because we had some naming conflicts and it was a lot of variable names to remember, which caused some confusion
    b. All buttons display slightly different columns so it was a challenge to deal with mapping data differently for each.
    c. We recommend that future teams should name variables more clearly, keep track of these variables, and implement normalization as it is good practice
4. Map Challenges (extracting additional data that we didn't initially plan for) - Ananya

a. It was a challenge to learn and figure out how to use the general syntax for Leaflet, which is the JS library for maps.
b. A big issue with this was that multiple listings would have the same location (if they were in the same building, for example), so then we couldn't tell if our map wasn't working or if some points were just conflicting with each other. This was why it was crucial to view the actual data (which we logged to the Network tab) rather than just relying on the visual points. Once we learned how to use console.log properly, it was much easier to debug and verify if the data was actually being pulled properly.
c. We recommend console logging (information for the coordinates, for example) because it helped us with debugging and seeing underlying issues.

Describe future work that you think, other than the interface, that the application can improve on.

Other than the interface, the application can improve on using more data from more locations, such as more states and more countries. Right now, we only have data for Airbnbs in Chicago, IL since that data itself was already 1000+ lines in our table in GCP. We can also think about incorporating flight and transportation costs instead of just the Airbnb prices. We also can think of ways to include real time ticket pricing, since right now we also don't consider ticket pricing since we don't have access to TicketMaster's data for that. We can also improve by making our application be unique to each user and having personalized recommendations. For example, notifying users when their favorite artists announces new shows, and recommending events based on user listening habits.

Describe the final division of labor and how well you managed teamwork.

Each team member contributed to a distinct portion of the project while still collaborating during integration. Michelle focused on gathering and cleaning the Ticketmaster data, helping prepare it into CSVs, and assisting with uploading and structuring the data in GCP. Ashley took responsibility for collecting and cleaning Airbnb datasets, contributing to the database schema design, normalizing the data in the tables, and helping populate the tables. Ananya worked primarily on the backend, including building the endpoints and debugging the database connection. We also all worked on implementing the advanced SQL queries in the backend, and connecting the backend to the GCP database. Ana handled the frontend, creating the search interface and the map and helped ensure that backend data successfully rendered in the application. Overall, teamwork was well-managed: everyone fulfilled their core responsibilities while supporting one another during debugging, testing, and database integration, which made the workflow smooth and balanced.

Main Changes

Discuss if you changed the schema or source of the data for your application
We did not change the source of data for our application, as we just decided to use Ticketmaster and Airbnb data as described in our proposal.

Discuss what you change to your ER diagram and/or your table implementations. What are some differences between the original design and the final design? Why? What do you think is a more suitable design?
Our final implementation preserved the core entities and relationships from the original ER diagram but introduced several practical changes as we integrated real Ticketmaster and Airbnb data. Key adjustments included introducing auto-incrementing primary keys where needed, updating certain ID fields to BIGINT to adapt to the appropriate types, and adapting foreign key constraints to align with the structure of real-world data. We also expanded the WantsToAttend bridge table by adding a housing_confirmed column, allowing the system to record which Airbnb listing a user selects for each event. This change supports a fuller trip-planning workflow beyond simply marking interest in an event. Overall, the original ERD offered a solid theoretical foundation, but the final design is more suitable for our functioning application.

Discuss what functionalities you added or removed. Why?
Several functionalities from our original proposal were modified or removed after working with real Ticketmaster and Airbnb data. We initially planned to incorporate ticket prices into our calculations, but Ticketmaster's dataset did not include consistent pricing fields due to constantly changing real-time prices, so we removed that field. We also could not support artist-name searches because the API data was event-oriented rather than concert-specific, and performer names were not included. Similarly, we removed our state filter because our dataset only contained events from Chicago, though we kept distance and date filtering. Our "Surprise Me" feature also shifted from randomizing by genre to simply returning a random event, since Ticketmaster's CSVs did not include genre information. However, we added new functionality not in the original proposal. By introducing a housing column in the WantsToAttend table, users can now track if they've secured housing for each event that they want to attend. These adjustments allowed us to focus on functionalities that were actually supported by the data we had, ensuring that our final features were both usable and accurate.

Are there other things that changed comparing the final application with the original proposal?
Several design and UI elements also changed as we adapted to real data constraints. Most importantly, our interactive map was simplified to display only the selected event's venue and the top five closest or most relevant Airbnb listings. This change fit more naturally with the query results and provided a clearer, more meaningful visualization for users. Additionally, our overall user flow became more event-name-driven rather than artist-driven, reflecting the

limitations of the Ticketmaster dataset. While these changes shifted our application slightly away from its original concert-focused vision, the final product is more realistic and aligned with the structure of the data available to us.

Advanced database design

Stored Procedure

Implementation: sp_event_airbnb_summary()

Application Utility:

- Our stored procedure powers Endpoints #1 (Cheapest Airbnb per Event within a mile) and #3 (Events with Most Available Listings within 5 miles).
- Instead of running complex aggregation queries repeatedly from the application layer, we execute this procedure once to populate temporary tables with the cheapest Airbnb option per event (within 1 mile) and events ranked by number of available listings (within 5 miles)

Advanced Query Concepts:

- Multiple Joins: Combines Event, Venue, City, Nearby, and AirbnbListing tables
- GROUP BY Aggregation: Uses MIN(), COUNT(), AVG(), and ROUND()
- Subqueries: The procedure creates temporary result sets that are queried by the application

Why It's Valuable:

- It encapsulates complex logic in one place
- Runs all SQL inside database so we don’t need to send multiple queries

Trigger

Implementation: prevent_double_booking trigger on WantsToAttend

Event-Condition-Action:

- Event: BEFORE INSERT on WantsToAttend
- Condition: IF EXISTS a conflicting event on the same date for the same user
- Action: SIGNAL error to prevent the insert

Application Utility:

- Concert-goers can't physically attend two events on the same date.
- Our trigger enforces this business rule at the database level, preventing scheduling conflicts.
- When users try to add an event via the "Add Event to Attend" feature, the trigger automatically checks for date conflicts and blocks invalid bookings with a clear error message: "You already have an event scheduled on this date."

Why It's Database-Appropriate:

- The constraint spans multiple rows (requires checking other user events)
- It provides immediate feedback to prevent bad data

Transaction

Implementation: bulk-add-city endpoint with db.beginTransaction()

Advanced Query Concepts:

- Query 1 - Find Events: JOIN + ORDER BY + LIMIT
    - Joins Event, Venue, and City tables
    - Orders by date to find the 5 soonest upcoming events
- Query 2 - Check Conflicts: JOIN + GROUP BY aggregation
    - Joins WantsToAttend with Event
    - Groups by event_id and date to identify existing user commitments

Isolation & Application Utility: This transaction implements the "Add 5 Upcoming Events from City" feature, which is complex because it must:

- Find upcoming events in a city
- Check for date conflicts with the user's existing schedule
- Check for duplicate event IDs already in the user's list
- Insert only non-conflicting events as a single atomic operation

Why Transactional Consistency Matters:

- The transaction ensures that if any part fails (conflict detected, database error), the entire operation rolls back, maintaining data integrity.
- This is critical for a user-facing feature where partial failures would create a poor experience.

Constraints

Primary Keys:

- All tables have appropriate primary keys (User.user_id, Event.event_id, etc.)
- Composite primary keys for bridge tables (WantsToAttend, Nearby)

Foreign Keys:

- Enforce referential integrity across all relationships
- Venue → City, Event → Venue, Nearby → Event & AirbnbListing
- WantsToAttend → User & Event

Additional Constraints:

- UNIQUE(email) on User table prevents duplicate accounts
- NOT NULL constraints on critical fields like event names, venue names, city names
- availability_365 > 0 filter in queries ensures we only show bookable Airbnbs

Application Value:

- These constraints ensure data quality, which is essential for a travel planning app.
- Invalid venue-city relationships or broken event-venue links would render our cost calculations meaningless.
- The constraints guarantee that every event has a valid location, every Airbnb listing belongs to a real city, and every "wants to attend" entry references actual users and events.