Databases offer newer and better tools for enforcing consistency, but ORM's frequently ignore the consistency tools offered by databases. When you create a user with a unique username in Rails, for example, ActiveRecord will make four database queries, to start a transaction, check whether any users in the database have that username, insert the user, and commit.

This is slow - ActiveRecord makes four roundtrips - and incorrect under READ COMMITTED, as Bailis et al discuss in "Feral Concurrency." Bailis and your database analyst would suggest using a unique index. Try the INSERT, let the database report failure and then bubble that up to the end user. Why don't ORM's like ActiveRecord do that? How can we help them use the features of the database?

In some ways ActiveRecord is a victim of its large, potentially oblivious userbase. Imagine that ActiveRecord wanted to depend on the unique index and just "try the write." If a user does not use ActiveRecord to run migrations, and forgets to add the unique index, ActiveRecord will always report that the INSERT succeeded, and she'll end up with duplicate records.

ActiveRecord works with many different datastores. When the library does a SELECT before INSERT and finds a matching row, it can return its own error class. By contrast, if they caught an error coming back from the database, now they have to parse a MySQLError, a PostgresError, a SqliteError, and find and compare the correct error code for each one. This adds complexity to error handling.

Shyp, an on-demand logistics company with $62 million in venture capital funding, managed to dramatically reduce the rate of oncall incidents by using Postgres correctly. Shyp uses Waterline, a Javascript ORM with no transaction support. Half of Shyp's oncall incidents were drivers or pickups getting "stuck" — two drivers assigned to the same pickup, for example. Implementing database transactions for driver assignment drastically reduced the incidence of stuck pickups and drivers, and the oncall load.

Another time, an error in the driver app caused packages to be re-acquired by drivers after they had been delivered to the warehouse. Shyp fixed this problem by introducing a state machine - an UPDATE with a WHERE clause that could be used as part of a transaction. Shyp also removed instances of `object.save()` from the codebase, which clobbered all columns on the database row.

These changes required significant work. When Waterline bubbled up a Postgres constraint failure, the test runner exited with 1, but attached no error message, stack trace or context. We had to build new database access libraries, and train the team to think first about the data layout, constraints, and query patterns required in a new feature, and work backwards to the application logic.

How can we help application users and ORM's use the features of the database? Some ORM's like SQLAlchemy do the right thing with constraints, and we should study their design. We could consider introducing a standardized set of error messages and error codes for common errors. Why should authors have to know that a uniqueness failure is Postgres error 23505, and MySQL error 1169? With some guidance, we can help ORM authors (and their end users) achieve much better concurrency, and help their oncall teams sleep easier.