dbt Code Standards for Large Organizations

With 30 analysts contributing to a DBT project, maintaining code **quality**, **clarity** and **documentation** becomes extremely important.

There are standards and best practices that should be implemented in a group like this, and I'd break those down into **three key pillars**:

1. **Code Quality**: This includes SQL style guides and naming conventions. For example, we want consistent patterns for model names, clear documentation, and readable code. As a software engineer as well as a data engineer, I've always been a fan of the book "Clean Code" by Robert C. Martin. One of the things that particularly resonates with me is the concept of readable code. What that translates to in this situation is functions should be short and easy to understand with minimal documentation, if any. Variable names should be self-explanatory, and the Single Responsibility Principle (SRP) should be enforced. I've found that clean, readable, well-structured code saves countless hours of debugging and handoffs between team members. This is usually a work in progress and takes a little time to get used to in a team where analysts haven't been exposed to principles of clean code; so I would recommend that every time a commit is made into GitHub, at least 2 other members of the team, including a senior one responsible for enforcing new rules, approve it before merging. If this is an existing team and we're enforcing new rules, they should be introduced in stages, so as to not overwhelm the analysts.

2. **Performance Standards**: Since we're using Snowflake, we need to be smart about how transformations are written. This means optimizing joins, leveraging Snowflake-specific features like clustering where it makes sense, and being mindful of expensive operations on large datasets.

3. **Testing Requirements**: Every model needs proper testing. This includes checking for null values, ensuring unique keys, validating referential integrity, etc. These aren't just nice-to-haves - they're essential for data quality. dbt has basic tests that are easy to implement for null values and uniqueness, and more complex custom tests are also available.

Automated enforcement is key for any team, especially a large one. We should make it impossible to merge non-compliant code, while not creating bottlenecks for the team. My approach would follow this process:

First, I'd set up SQLFluff as a SQL linter. It's fantastic for dbt projects because it can enforce style guides automatically. As mentioned earlier, we'll start with a minimal set of rules and

gradually expand them; Enforcing everything at once will overwhelm the team, and create resistance to the project.

For the actual automated implementation, we would need to create a CI/CD pipeline using **GitHub Actions**, or another tool like **CircleCI** that runs on every pull request.

When an analyst pushes code the following happens:

- Pre-commit hooks run SQLFluff locally, catching basic issues before they even hit GitHub
- When they create a PR, GitHub Actions / CircleCI automatically:
    - Runs all DBT tests
    - Performs a full DBT build to catch any compilation issues
    - Executes SQLFluff checks
    - Validates documentation completeness

Something I've found particularly effective is **integrating this with Slack** and/or email. When tests fail, the analyst gets an immediate notification with details about what went wrong. This speeds up the feedback loop significantly.

For orchestration, I'd use Prefect or Airflow to schedule these checks and maintain a record of test results. This becomes invaluable when you need to track quality trends over time. I'd also implement a 'quality score' dashboard that tracks metrics like test coverage, documentation completeness, and failed checks per PR. This will give visibility into where we need to focus our improvement efforts.

Another important aspect is team buy-in. I've found that running workshops to explain why these standards matter and how they make everyone's life easier in the long run really helps with team-wide adoption. It may be helpful to set up regular 'DBT office hours' where analysts can get help with optimizing their code or understanding test failures. If there is a regular internal email that goes out or an engineering blog entry that is regularly posted, this might be a good place to highlight what we’re trying to accomplish and explain why adoption is good for the team.

The key to making all this work is finding the right balance between strictness and productivity. We want the team to feel empowered, and we want to set up guardrails, not roadblocks.