***Gates***

* **Database Model** - Build tables that balance the number of joins (normalization) versus a reasonable amount of denormalization.  If you want the application to be highly normalized for the opportunity to extend it for more generic needs, be sure you understand the trade-off of having to join a potentially large number of tables.  On the other side of the coin, if your database only has a handful of tables that are 200+ columns with a great deal of duplicate data, you may want to strike a balance.
* **Primary Keys** - Ensure a column in each table is designated as the primary key and select an appropriate data type such as an integer.
* **Foreign Keys** - Validate all of the foreign keys are established to maintain referential integrity among your tables to ensure the proper relationships are maintained.
* **Auto Increment** - If a column needs an incremental value, use an identity for that column as opposed to a custom solution.
* **Correct Data Types** - Be sure to select the correct data types for each column. For example, if a varchar column can be used as opposed to a BLOB data type, in general the performance implications will be far less.  So choose the data type wisely.
* **Indexing** - As new queries are written, columns are added or removed, it is a good time to validate that your indexes will support your performance needs.  Validate indexes are not unnecessarily duplicated and validate as indexes are added that the intended queries take advantage of the query.  If not, think twice about building a new index that SQL Server will need to use, but your new query is not using.
* **Statistics** - Validate sufficient statistics are available on a per column basis or validate based on the database configurations that the indexes will be created.
* **Lookup Tables** - As opposed to being forced to hard code values in T-SQL code, lookup the values in a table with the proper referential integrity.  Over time the code will be much easier to understand and simpler to maintain.
* **Defaults** - As applicable, create defaults for the columns to have a meaningful value for the column.
* **NULL Usage** - Review the NULL usage on a per-column basis and determine if NULL is an applicable value or if a business specific value will be used instead which can be enforced via a lookup table.
* **Testing Results** - Validate the functional testing was completed successfully to ensure you will not face functional issues once the code is deployed and be in a situation where you are asked to change the code on the fly.
* **Response Time Requirements** - Validate the performance testing yielded acceptable performance for the new code that is going to be released as well as not negatively impact the existing platform.  To me, the only situation worse than having to change code on the fly is the need to firefight a performance issue.
* **Scope of Data** - Only process (SELECT, INSERT, UPDATE or DELETE) the needed data. I have seen in a few different applications where SQL Server is flooded with SELECT statements that return unneeded data that is unnecessarily filtered by the front end application. A much more efficient process is to just select the data needed. On another occasion, I have observed duplicate data being unnecessarily inserted into the table. The resolution in this specific scenario is duplicate data that needs to be filtered on the front end.

***Stored Procedure Considerations***

* **General Review** - As you review the code, make sure you do not see any of the following:
  + Unnecessary JOINs
  + Inaccurate calculations or functions
  + Unnecessary data
  + Cartesian product
* **General Technique** - Make sure the general coding practice is reasonable.  If possible, use SET based logic as opposed to a cursor or a while statement. In addition, avoid logic that processes data in a temp table as opposed to processing the data directly.  For example, there is no reason to load data into a temporary table, update the data in the temporary table, delete the data from the base table and then load the data from the temporary table.  Just update the needed data in the base table.
* **Query Plan** - Review the query plan to validate it is efficient.  Validate the following items:
  + Table scanning is not occurring
  + Bookmark lookups are not occurring
  + Code or indexes can be changed to convert index scans to index seeks
* **Optimizer Overrides** - Review the code to determine if  index hints or NOLOCK clauses are really necessary or if a portion of the code can be changed to not try to out smart the optimizer.  These hints could be beneficial in the short term, but as the database or code changes as well as when new versions of SQL Server are released, these overrides can be an unexpected exercise when it comes time to upgrade.
* **Comments** - Be sure valuable comments are included with the code; if not, it is rare that comments are added once the code is deployed.

***Rollback Considerations***

In general, one of the big items that is overlooked when it comes to deploying code is a rollback plan.  You should always plan for the worst and be prepared if a problem does arise.  Too often an issue does arise and then as a DBA, you are in fire fighting mode to resolve the issue.

* **Inverse Code** - As you deploy code, be sure to have code to rollback the code deployment.  For example, if you have CREATE code, have the corresponding DROP code. If you are ALTERing an object, have the original code to roll back to that version.
* **Testing** - Test the code to validate that it parses correctly and will rollback the code if an unforeseen issue arises.
* **Record Error** - If you deploy the code and face an unexpected issue, record the error in order to troubleshoot the issue and share this knowledge among your team members.
* **Rollback** - If an issue arises and it is not something that cannot be easily changed, then rollback the code to the original state.