Beyond Generic Tests

Valuable dbt Testing Techniques

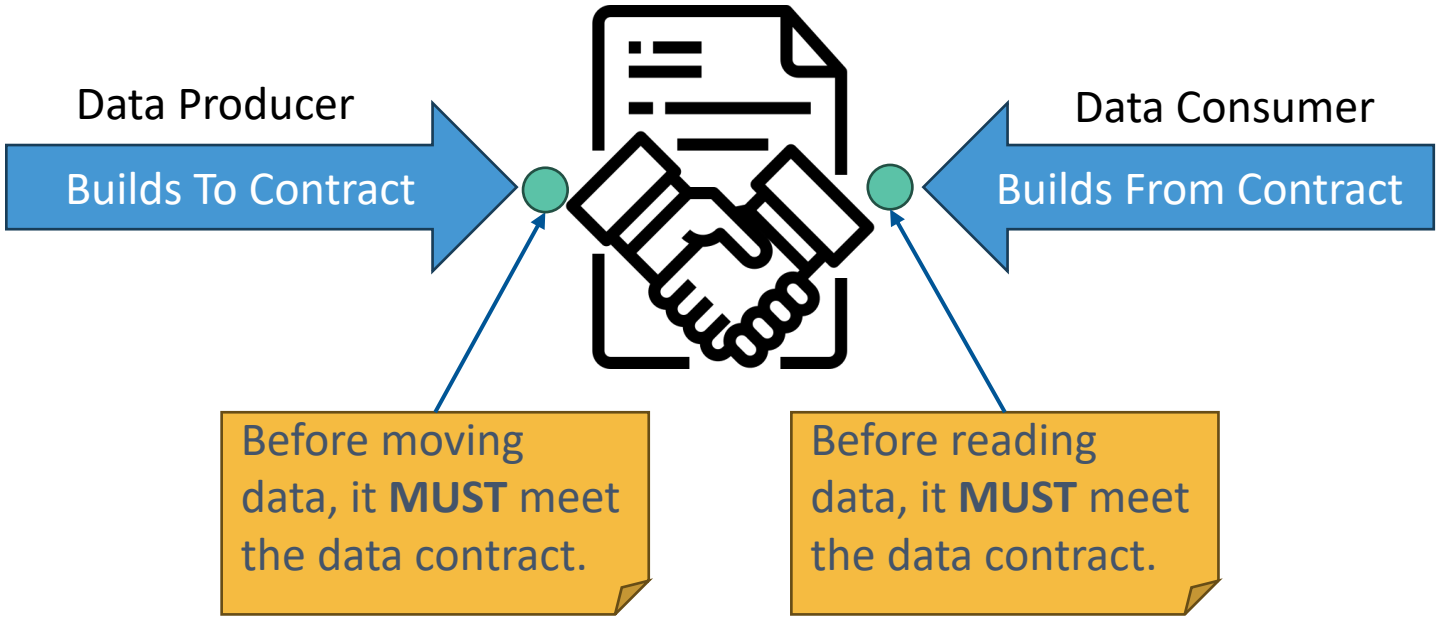
Topics Covered

1. Testing Types: Demystified
2. dbt Testing Features & Packages
3. Data Contract Validation (source & target)
4. Data Quality
5. Functionality: Unit Tests
6. Implementing a model & testing

Testing Types: Demystified

Understand the types of tests you'll encounter

Data Contract Validation



What's in a Contract?

There is no real standard, but could include:

- Exchange format (json, yaml, csv, etc.)
- Data structure name(s)
- Schema(s)
- Data types
- *Load timing/frequency*
- *Security/access*
- *…and more*

Data Contracts 101 (Monte Carlo)

Icon by itim2101

Data Quality Validation

Measurement of the condition of data

https://www.stmarys.ac.uk/planning/data-quality.aspx

improving

Functional Validation

Ensure software behavior operates as expected

Given the state of a system (inputs), when the software is executed, is the output correct?

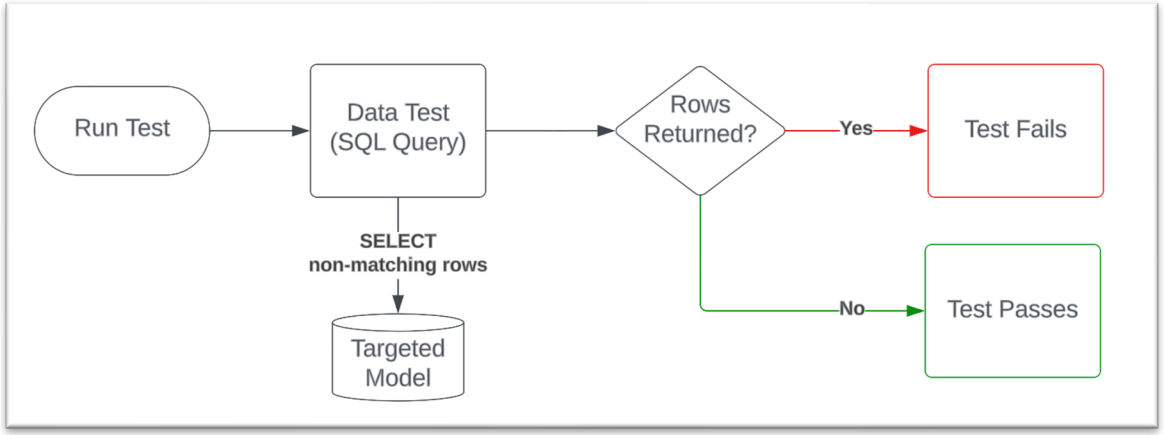
- Unit – a specific module or function
- Integration – multiple modules and/or system integrations
- Performance – speed of processing
- Load – behavior under load
- Security – Access control

Automation enables the activity of cheap, repeatable regression testing of functionality.

dbt Testing Features & Packages

An overview of dbt tests and helpful packages

dbt Testing Features



Example:
`not_null` test of `code` column on `silver.carrier`

```
SELECT * FROM silver.carrier WHERE code IS NULL
```

Types of tests in dbt (typically data testing)

- Generic tests (unique, not_null, accepted_values, relationship)
- Singular tests (custom tests written in SQL)
  - Formerly known as bespoke tests

Executing tests in dbt

- dbt test
- dbt test --select [model, lineage, tags, or specific tests]
- dbt build (runs pipeline and tests)

Useful Packages to Improve Quality

dbt-expectations

https://github.com/calogica/dbt-expectations

- Port of popular python data quality tool, Great Expectations, into dbt tests
- Advanced data quality testing features
- Configurable using dbt test behavior in[model].yaml files

dbt-unit-testing

https://github.com/EqualExperts/dbt-unit-testing

- Adds **functional** unit testing capabilities for a model
- Tests model behaviors, not data condition
- Mock data inputs and validate against expected outputs

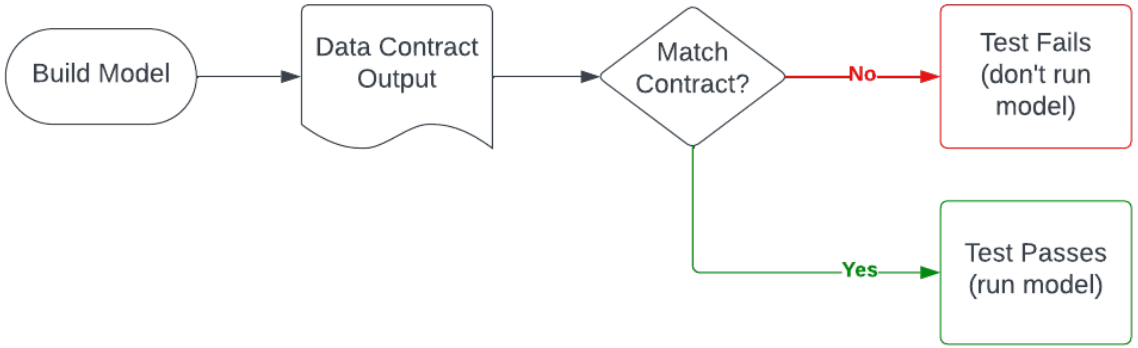
Honorable Mention: Audit Helper
https://github.com/dbt-labs/dbt-audit-helper/tree/0.9.0/

Data Contract Validation

Do the sources and your target have the right data contracts?

Data Contract Testing (Model Contracts)

New in dbt 1.5.x: https://docs.getdbt.com/docs/collaborate/govern/model-contracts




```yaml
models:
  - name: gld_airport
    alias: airport
    config:
      contract:
        enforced: true
    columns:
      - name: code
        data_type: character varying (8)
        tests: …
      - name: name
        data_type: character varying (128)
        tests: …
      - name: city
        data_type: character varying (128)
        tests: …
      - name: state
        data_type: character varying (32)
        tests: …
```

Does the model build to this contract?

```sql
{{ config(materialized='view', alias='airport')}}

SELECT
    code,
    name,
    CAST(city AS CHARACTER VARYING(128)),
    CAST(state_country AS CHARACTER VARYING (32)) AS state
FROM {{ ref('slv_airport')}}
WHERE
    LENGTH(state_country) = 2
    AND city IS NOT NULL
    AND name IS NOT NULL
```

Model Contract Validation

Run-time check of model output schema

Model Configuration

```yaml
version: 2

models:
  - name: slv_airport
    config:
      contract:
        enforced: true
    columns:
      - name: code
        data_type: character varying(8)
      - name: name
        data_type: character varying(128)
      - name: city  # Need a test for "does not have a comma"
        data_type: character varying(32)
      - name: state_country
        data_type: character varying(8) # NOTE: length is NOT ENFORCED
```

FAILING Model Definition

```sql
{{ config(materialized='view', schema='silver', alias='airport')}}

SELECT
    code,
    ''-- as city,
    '',-- AS state_country,
    description AS name
FROM {{ source('demo_source', 'airport_code')}}
```

PASSING Model Definition

```sql
{{ config(materialized='view', schema='silver', alias='airport')}}

SELECT
    code,
    '' as city,
    '' AS state_country,
    description AS name
FROM {{ source('demo_source', 'airport_code')}}
```

Data Contract Pre-Validation

Validate sources/dependencies for a model are correct first

Before running a model, ensure that the contract of dependent models are correct.

Options:

- Assign the checks and always run them as a `dbt test` call before running `dbt build`
- Tag the tests, e.g. 'precheck' and run as `dbt test --select tag:precheck`

expect_table_columns_to_match_ordered_list

Expect the columns to exactly match a specified list.

Applies to: Model, Seed, Source

```yaml
models: # or seeds:
  - name: my_model
    tests:
      - dbt_expectations.expect_table_columns_to_match_ordered_list:
          column_list: ["col_a", "col_b"]
          transform: upper # (Optional)
```

expect_column_values_to_be_of_type

Expect a column to be of a specified data type.

Applies to: Column

```yaml
tests:
  - dbt_expectations.expect_column_values_to_be_of_type:
      column_type: date
```

expect_column_to_exist

Expect the specified column to exist.

Applies to: Column

```yaml
tests:
  - dbt_expectations.expect_column_to_exist
```

expect_column_values_to_be_in_type_list

Expect a column to be one of a specified type list.

Applies to: Column

```yaml
tests:
  - dbt_expectations.expect_column_values_to_be_in_type_list:
      column_type_list: [date, datetime]
```

https://github.com/calogica/dbt-expectations

expect_table_columns_to_match_set

Expect the columns in a model to match a given list.

Applies to: Model, Seed, Source

```yaml
models: # or seeds:
  - name: my_model
    tests:
      - dbt_expectations.expect_table_columns_to_match_set:
          column_list: ["col_a", "col_b"]
          transform: upper # (Optional)
```

expect_table_column_count_to_equal

Expect the number of columns in a model to be equal to `expected_number_of_columns`.

Applies to: Model, Seed, Source

```yaml
models: # or seeds:
  - name: my_model
    tests:
      - dbt_expectations.expect_table_column_count_to_equal:
          value: 7
```

expect_table_columns_to_contain_set

Expect the columns in a model to contain a given list.

Applies to: Model, Seed, Source

```yaml
models: # or seeds:
  - name: my_model
    tests:
      - dbt_expectations.expect_table_columns_to_contain_set:
          column_list: ["col_a", "col_b"]
          transform: upper # (Optional)
```

Table & Column-Level Validations

Validate columns expected, test with a precheck tag

```yaml
version: 2

sources:
  - name: demo_source
    schema: 'bronze'
    tags: ['precheck']
    tables:
      Generate model
      - name: airport_code
        tests:
          - dbt_expectations.expect_table_columns_to_match_ordered_list:
              column_list: ["code", "description"]
              transform: upper # (Optional)
      Generate model
      - name: carrier_code
      Generate model
      - name: flight_data
```

```yaml
version: 2

sources:
  - name: demo_source
    schema: 'bronze'
    tables:
      Generate model
      - name: airport_code
        columns:
          - name: code
            tests:
              - dbt_expectations.expect_column_to_exist:
                  tags: ['precheck']
          - name: description
            tests:
              - dbt_expectations.expect_column_to_exist:
                  tags: ['precheck']
      Generate model
      - name: carrier_code
      Generate model
      - name: flight_data
```

NOTE:

Tags can be applied at many levels, including a specific test itself
https://docs.getdbt.com/reference/resource-configs/tags#other-resource-types

Data Quality

Testing of various dimensions of data quality

Basic Data Quality

Understand the basic data quality constraints of your contract

```yaml
version: 2

models:
  - name: slv_airport
    config:
      contract:
        enforced: true
    tests:
      - dbt_expectations.expect_table_row_count_to_equal_other_table:
          compare_model: source('demo_source', 'airport_code')
          compare_row_condition: "code != 'ZZZ'"
    columns:
      - name: code
        data_type: character varying(8)
        tests:
          - not_null
          - unique
          - dbt_expectations.expect_column_values_to_not_be_in_set:
              value_set: ['ZZZ', 'zzz']
              quote_values: true # (Optional. Default is 'true'.)
      - name: name
        data_type: character varying(128)
        tests:
          - not_null
      - name: city  # Need a test for "does not have a comma"
        data_type: character varying(32)
      - name: state_country
        data_type: character varying(8) # NOTE: length is NOT ENFORCED
```

Table is a straight pull, except for 'ZZZ' codes

code:

- cannot be null
- it's a key, so must be unique
- Value can never have 'ZZZ'

Where should not_null be tested?
On the source before the pipeline runs, or after the data is loaded?

If done as a pre-check, you can avoid coding for nulls, but if someone modifies the table via another means, it'll slip by.

OPTION: Do both!

More DQ: Validate Data Types

Especially useful on views, ensure data types are as expected

```yaml
version: 2

models:
  - name: slv_airport
    config:
      contract:
        enforced: true
    tests:
      - dbt_expectations.expect_table_row_count_to_equal_other_table:
          compare_model: source('demo_source', 'airport_code')
          compare_row_condition: "code != 'ZZZ'"
    columns:
      - name: code
        data_type: character varying(8)
        tests:
          - not_null
          - unique
          - dbt_expectations.expect_column_values_to_not_be_in_set:
              value_set: ['ZZZ', 'zzz']
              quote_values: true # (Optional. Default is 'true'.)
          - dbt_expectations.expect_column_value_lengths_to_be_between:
              min_value: 1
              strictly: false
          - dbt_expectations.expect_column_values_to_be_of_type:
              column_type: character varying
      - name: name
        data_type: character varying(128)
        tests:
          - not_null
          - dbt_expectations.expect_column_values_to_be_in_type_list:
              column_type_list: [character varying, text]
```

- Must have at least one character
- Must be a character varying

- Could be character or text

improving

More DQ: Validate Data Content

Validate valid values using lists, regex, etc.

```yaml
      - name: name
        data_type: character varying(128)
        tests:
          - not_null
          - dbt_expectations.expect_column_values_to_match_regex:
              regex: "[[:alnum:]]+"
          - dbt_expectations.expect_column_values_to_be_in_type_list:
              column_type_list: [character varying, text]
      - name: city  # Need a unit test for "description does not have a comma"
        data_type: character varying(32)
        tests:
          - not_null
          - dbt_expectations.expect_column_values_to_match_regex:
              regex: "[[:alnum:]]+"
          - dbt_expectations.expect_column_values_to_be_in_type_list:
              column_type_list: [character varying, text]
```

- Must be alphanumeric and not empty

- Must be alphanumeric and not empty

improving

Other Favorite dbt-expectations tests

More configurable than dbt generic tests

expect_column_values_to_be_in_set

Expect each column value to be in a given set.

Applies to: Column

```
tests:
  - dbt_expectations.expect_column_values_to_be_in_set:
      value_set: ['a','b','c']
      quote_values: true # (Optional. Default is 'true'.)
      row_condition: "id is not null" # (Optional)
```

expect_column_values_to_be_between

Expect each column value to be between two values.

Applies to: Column

```
tests:
  - dbt_expectations.expect_column_values_to_be_between:
      min_value: 0  # (Optional)
      max_value: 10 # (Optional)
      row_condition: "id is not null" # (Optional)
      strictly: false # (Optional. Default is 'false'. Adds an 'or equal to' to the comparison opera
```

expect_column_values_to_not_be_in_set

Expect each column value not to be in a given set.

Applies to: Column

```
tests:
  - dbt_expectations.expect_column_values_to_not_be_in_set:
      value_set: ['e','f','g']
      quote_values: true # (Optional. Default is 'true'.)
      row_condition: "id is not null" # (Optional)
```

expect_column_values_to_be_unique

Expect each column value to be unique.

Applies to: Column

```
tests:
  - dbt_expectations.expect_column_values_to_be_unique:
      row_condition: "id is not null" # (Optional)
```

expect_column_values_to_not_be_null

Expect column values to not be null.

Applies to: Column

```
tests:
  - dbt_expectations.expect_column_values_to_not_be_null:
      row_condition: "id is not null" # (Optional)
```

expect_column_values_to_be_null

Expect column values to be null.

Applies to: Column

```
tests:
  - dbt_expectations.expect_column_values_to_be_null:
      row_condition: "id is not null" # (Optional)
```

DQ Checks Passed

Is my code correct?

Functionality: Unit Tests

Validating transformations will work correctly beyond the happy path

Define Unit Tests for a Model

dbt-unit-testing: https://github.com/EqualExperts/dbt-unit-testing

Problems with Data Tests:

- Reactive to data that has been processed
- Edge cases that could break processing may not be present
- Do not validate the functionality of a model, only the output

If you are writing code, especially to handle edge cases (case/when, cast, etc.), you need functional tests to:

- Ensure edge cases are handled, even if they aren't in the data
- Understand expected behavior of joins used (left, inner, etc.)

Anatomy of a Functional Test

GIVEN

Some state of the data (input data)

WHEN

A model is executed

THEN

The output rows should equal [some expected data]

Happy Path Unit Test

Unit testing gets run as part of the “test” step.
Prefer: dbt test --select tag:unit-test

```
{{
    config(
        tags=['unit-test', 'no-db-dependency']
    )
}}

{% call dbt_unit_testing.test('slv_airport', 'validates parsing logic') %}

    {% call dbt_unit_testing.mock_source ('demo_source','airport_code') %}
        CODE    | Description
        '01a'   | 'city1a, AL: 01A Airport'
        'ZZZ'   | 'this value should not matter'
    {% endcall %}

    {% call dbt_unit_testing.expect() %}
        CODE    | CITY      | STATE_COUNTRY                 | NAME
        '01a'   | 'city1a'  | 'AL'                          | '01A Airport'
    {% endcall %}
{% endcall %}
```

Model being tested

Happy path test input data

Expected output

NOTE:

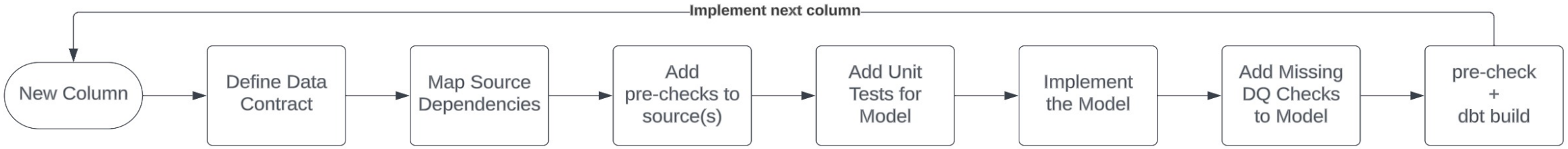
This is only testing a model that selects from a single model and outputs to a single model. You can mock and “expect” more than one of each.

Implementing a Model and Testing

Process for building tests into model development proactively

Development Process Overview

A method to integrate testing into the process



Iteratively build the model, one column at a time. Add the easy, obvious, and basic tests as you go.

Demo Use Case

Model Use Case:

Given a carrier table as input, parse it into the silver layer, cleaning up core data issues, while having a dataset that can be used for analytics or to create a gold data set.

Inputs:

bronze.carrier_code

- code column that has a carrier code
- description column that has carrier, and effective dates
  - Format: carrier name (fromYear – toYear)

Output Contract:

- Name: silver.carrier
- Columns
  - code – character varying (8)
  - name – character varying (128)
  - effectiveFromYear – integer
  - effectiveToYear – integer
  - originalDescription – character varying (128)

Carrier Parsing Logic

A few example data cases

BRONZE

code	description
01a	carrier 01a (2016 - 2020)
02a	carrier 02a (- 2020)
05a	carrier 05d

SILVER

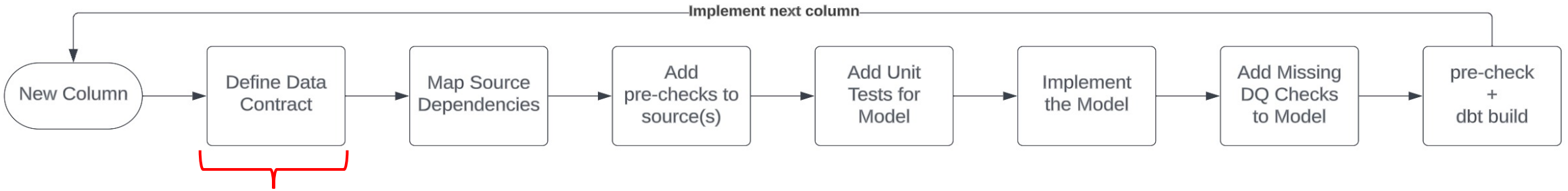
code	name	effectiveFromYear	effectiveToYear	originalDescription
01a	carrier 01a	2016	2020	carrier 01a (2016 - 2020)
02a	carrier 02a	**NULL**	2020	carrier 02a (- 2020)
05d	carrier 05d	**-1**	**-1**	carrier 05d

GOLD

code	name	effectiveFromYear	effectiveToYear
01a	carrier 01a	2016	2020
02a	carrier 02a	**1900**	2020

Implement: Carrier Code

Define Data Contract



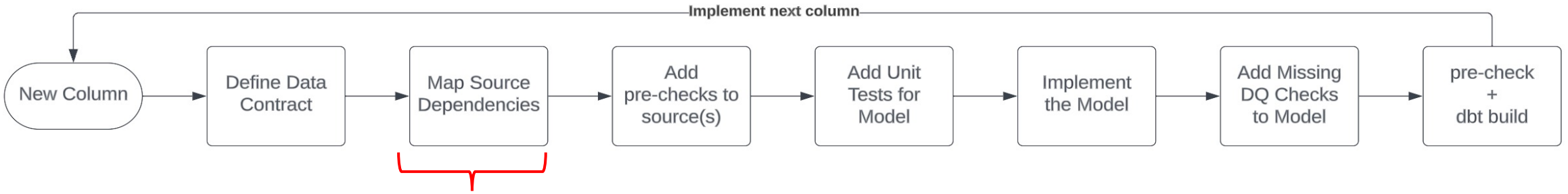
Your contract should map to the business insights desired, not “everything under the sun”.

The more unnecessary elements in the contract, the more code that’s required.

Field	Column Name	Data Type	Rules/Notes
Carrier Code	code	character varying (8)	Straight pull No nulls No Spaces

Implement: Carrier Code

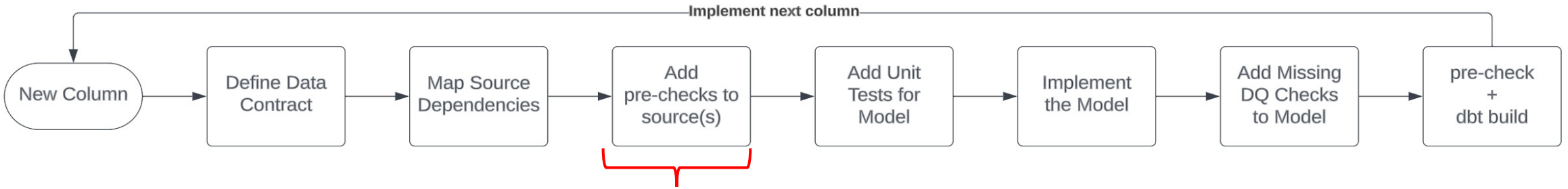
Map Source Dependencies



Field	Column Name	Data Type	Source Table	Source Field(s)	Rules/Notes
Carrier Code	code	character varying (8)	bronze.carrier_code	code	Straight pull No nulls No Spaces

Implement: Carrier Code

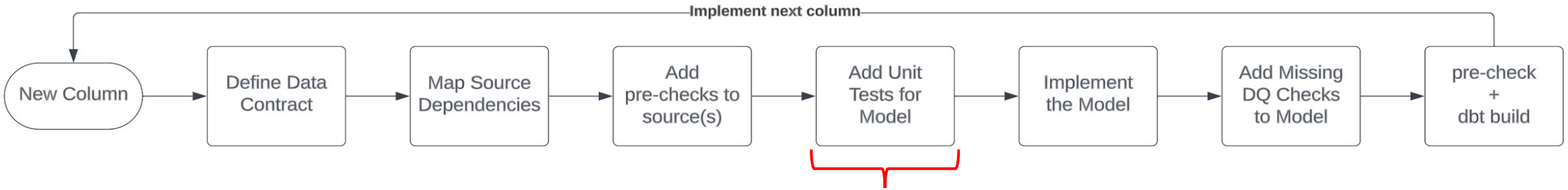
Add pre-checks to source(s)



Based on mapping, the source column should be validated as a pre-check

Implement: Carrier Code

Add Unit Tests

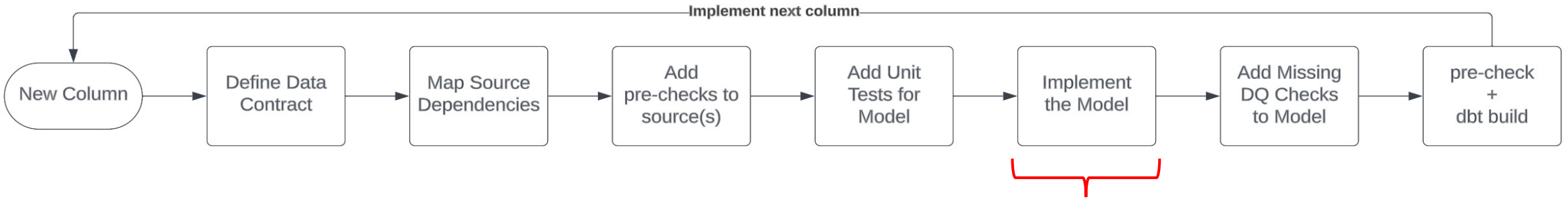


What are the various tests for the “code” field?

Focusing on one field allows for deeper, more focused thought

Implement: Carrier Code

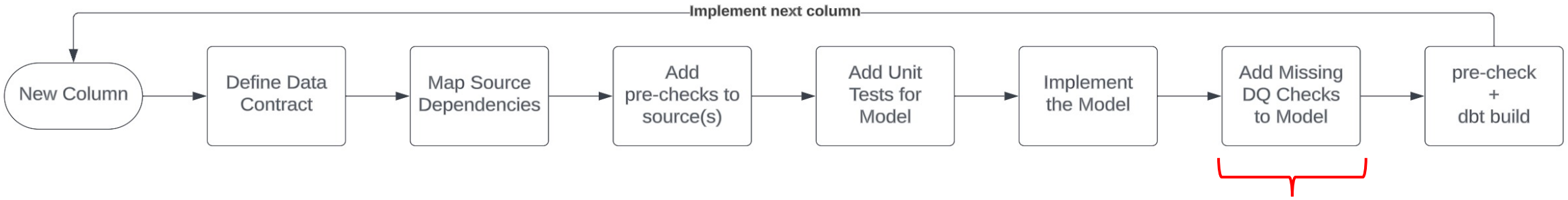
Implement the model



1. Implement the model
2. Run unit tests
3. Fix if needed

Implement: Carrier Code

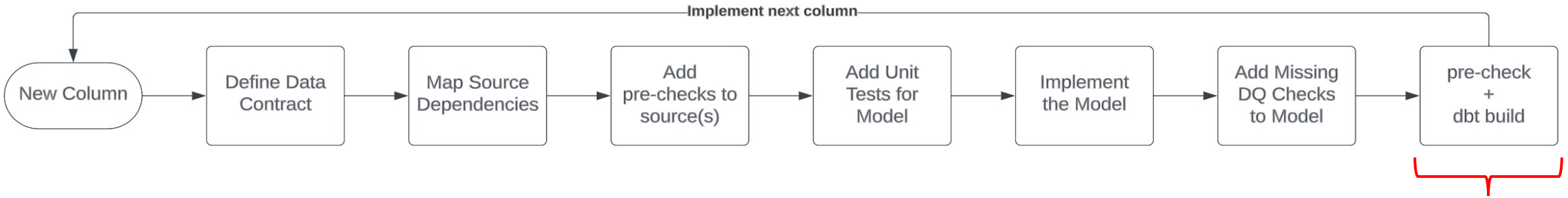
Add missing DQ check to model config



During coding, if some filters were applied or other DQ issues are discovered, add them NOW so you don't forget.

Implement: Carrier Code

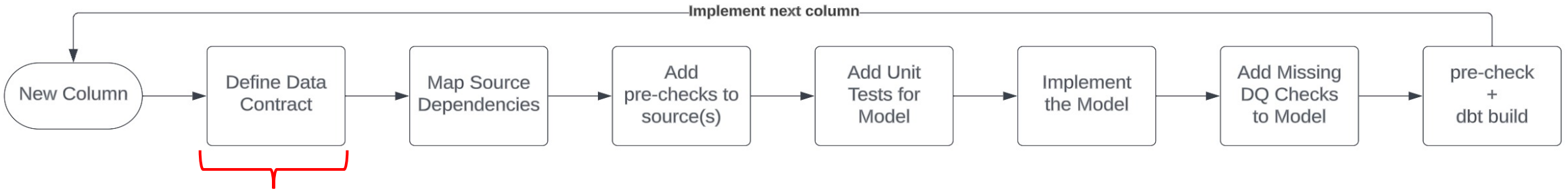
Regression test (test pre-check and dbt build)




```
dbt test --select tag:precheck
dbt build
```

Implement: Carrier Name

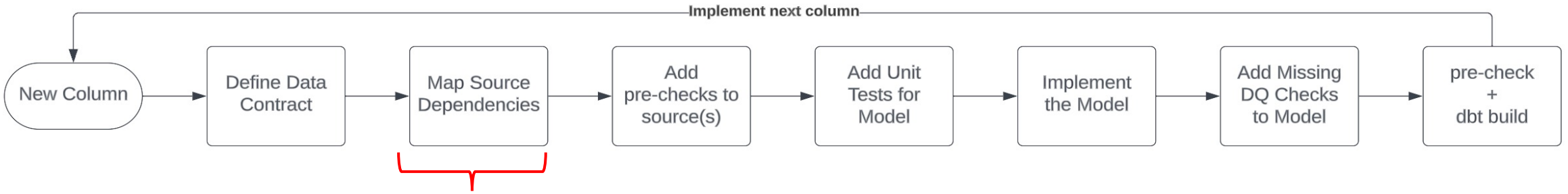
Define Data Contract



Field	Column Name	Data Type	Rules/Notes
Carrier Name	name	character varying (128)	string to the left of the last '(' Trim not null

Implement: Carrier Name

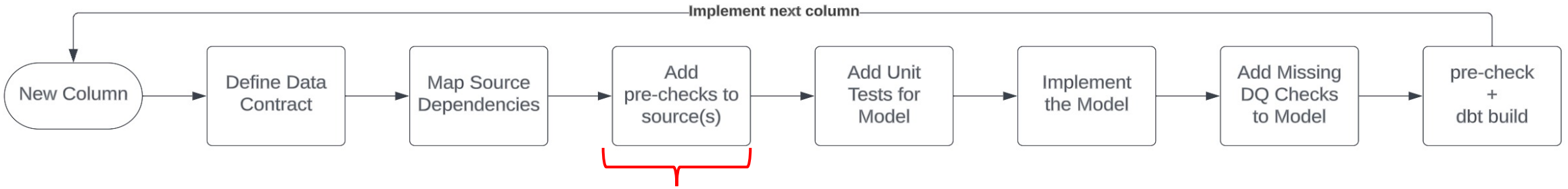
Map Source Dependencies



Field	Column Name	Data Type	Source Table	Source Field(s)	Rules/Notes
Carrier Name	name	character varying (128)	bronze.carrier_code	description	string to the left of the last '(' Trim not null

Implement: Carrier Name

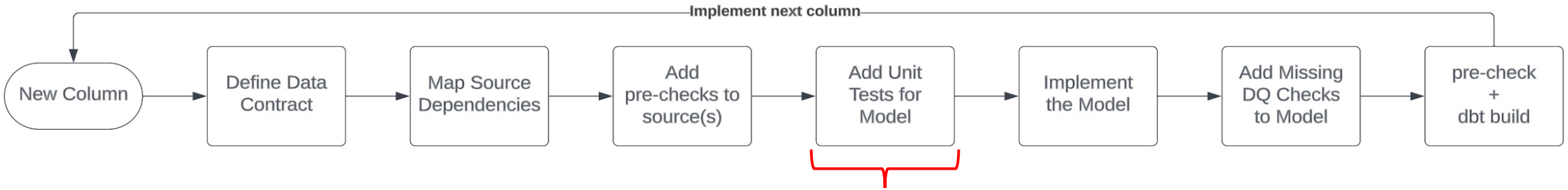
Add pre-checks to source(s)



improving

Implement: Carrier Name

Add Unit Tests

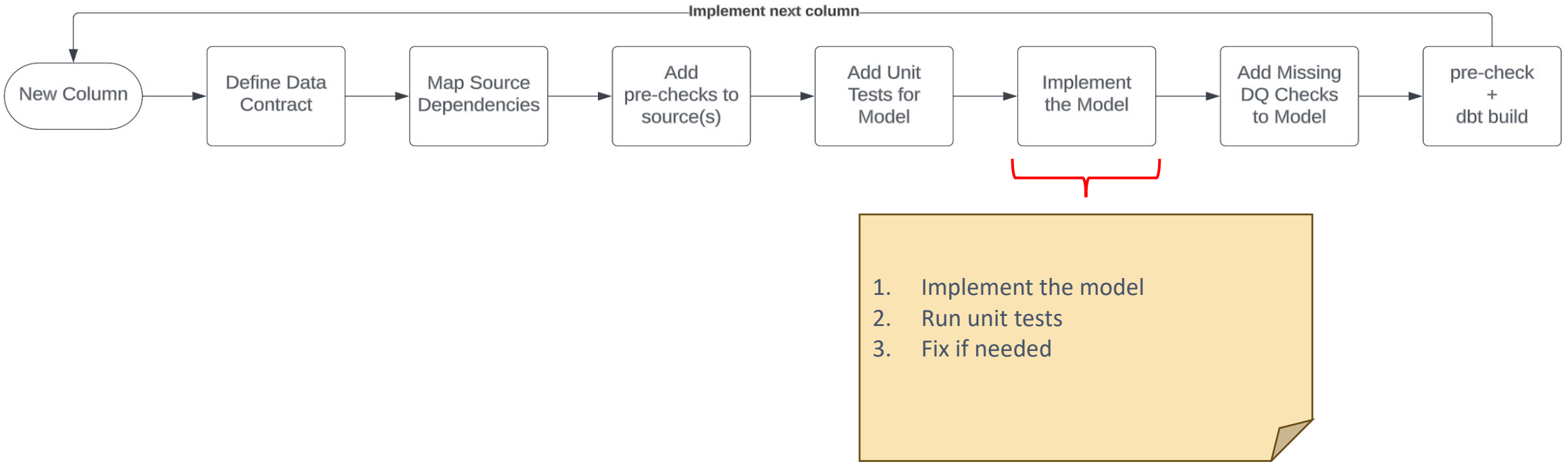


What are the various tests for the "name" field?

Focusing on one field allows for deeper, more focused thought

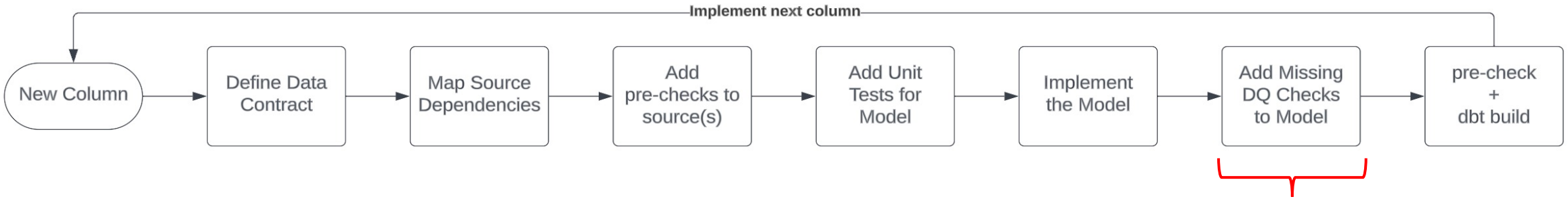
Implement: Carrier Name

Implement the model



Implement: Carrier Name

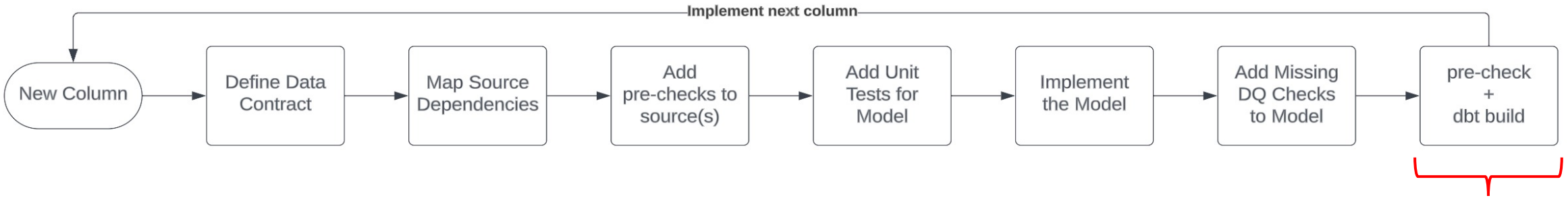
Add missing DQ checks to model config



During coding, if some filters were applied or other DQ issues are discovered, add them NOW so you don't forget.

Implement: Carrier Name

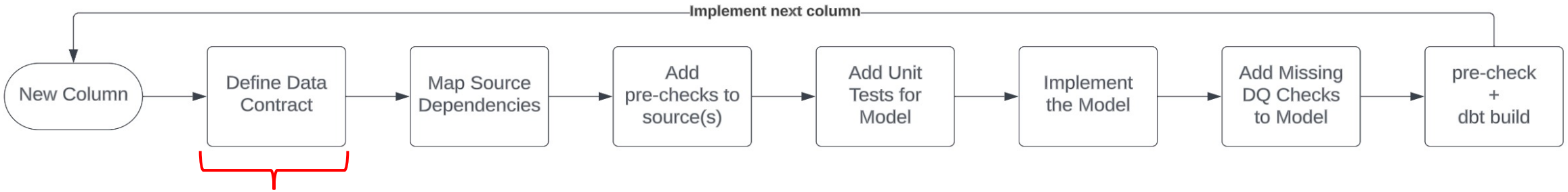
Regression test (test pre-check and dbt build)



dbt test --select tag:precheck
dbt build

Implement: Carrier Effective From Year

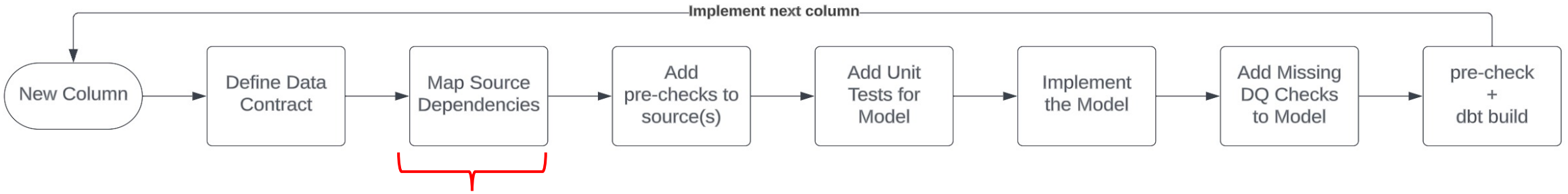
Define Data Contract



Field	Column Name	Data Type	Rules/Notes
Carrier name effective from year	effectiveFromYear	integer	first number in year range if missing, then null if can't parse, then -1

Implement: Carrier Effective From Year

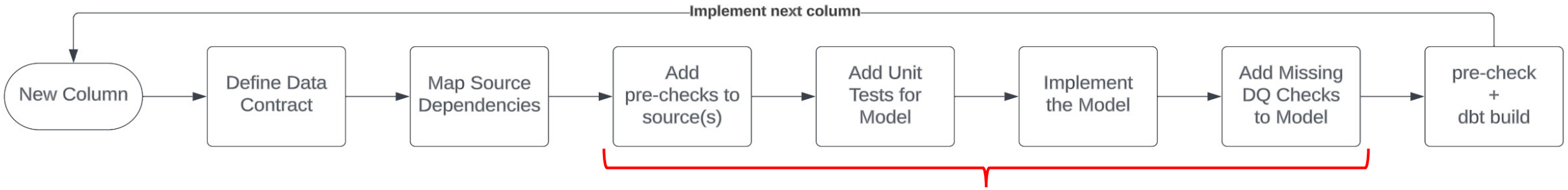
Map Source Dependencies



Field	Column Name	Data Type	Source Table	Source Field(s)	Rules/Notes
Carrier name effective from year	effectiveFromYear	integer	bronze.carrier_code	description	first number in year range if missing, then null if can't parse, then -1

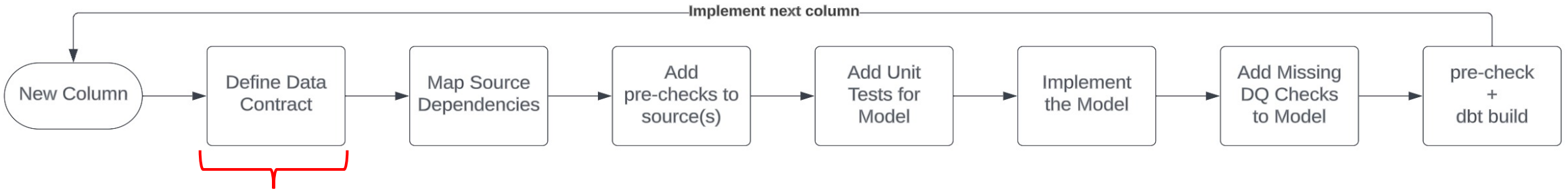
Implement: Carrier Effective From Year

Implement tests and model



Implement: Carrier Effective To Year

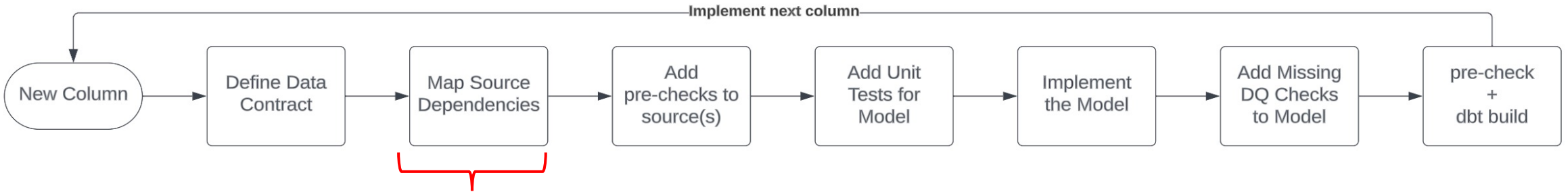
Define Data Contract



Field	Column Name	Data Type	Rules/Notes
Carrier name effective to year	effectiveToYear	integer	second number in year range if missing, then null if can't parse, then -1

Implement: Carrier Effective To Year

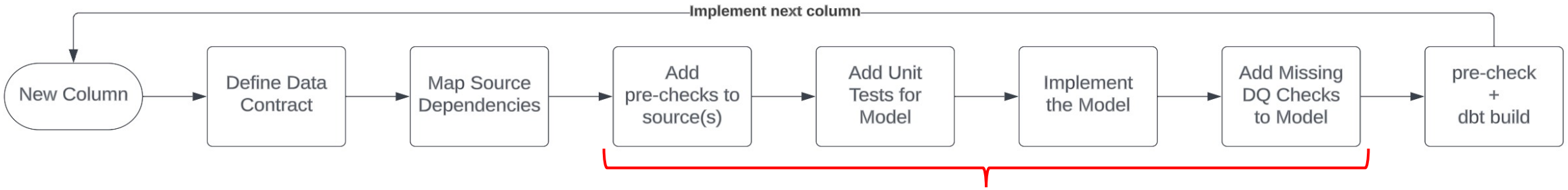
Map Source Dependencies



Field	Column Name	Data Type	Source Table	Source Field(s)	Rules/Notes
Carrier name effective to year	effectiveToYear	integer	bronze.carrier_code	description	second number in year range if missing, then null if can't parse, then -1

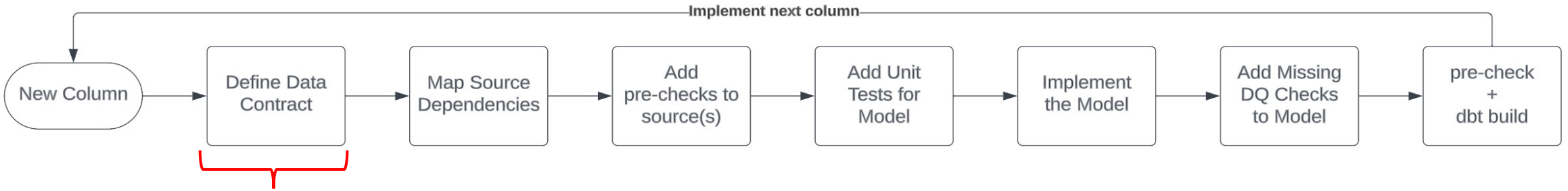
Implement: Carrier Effective To Year

Implement tests and model



Implement: Carrier Original Description

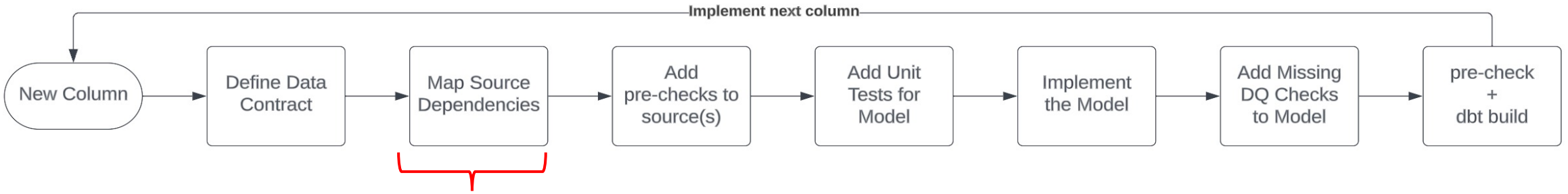
Define Data Contract



Field	Column Name	Data Type	Rules/Notes
Carrier original description	originalDescription	character varying (128)	straight copy of original description

Implement: Carrier Original Description

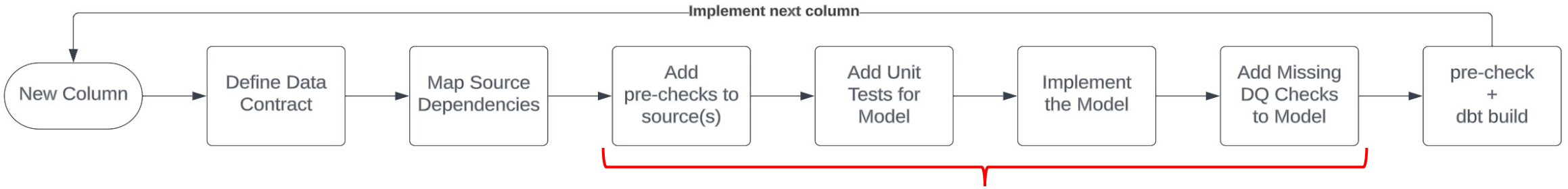
Map Source Dependencies



Field	Column Name	Data Type	Source Table	Source Field(s)	Rules/Notes
Carrier original description	originalDescription	character varying (128)	**bronze.carrier_code**	**description**	straight copy of original description

Implement: Carrier Original Description

Implement tests and model



SOOOO, Are We Done?

All that testing – it must be perfect!

D'OH!

Sometimes things slip through the cracks

code text	name text	effectivefromyear integer	effectivetoyear integer	originaldescription character varying (127)
02Q	Titan Airways	2006	[null]	Titan Airways (2006 -)
04Q	Tradewind Aviation	2006	[null]	Tradewind Aviation (2006 -)
05Q	Comlux Aviation, AG	2006	2012	Comlux Aviation, AG (2006 - 2012)
06Q	Master Top Linhas Aereas Ltd.	2007	[null]	Master Top Linhas Aereas Ltd. (2007 -)
07Q	Flair Airlines Ltd.	2007	[null]	Flair Airlines Ltd. (2007 -)
09Q	Swift Air, LLC	2006	2017	Swift Air, LLC (2006 - 2017)
09Q	Swift Air, LLC d/b/a Eastern Air Lines d/b/a Eastern	2018	[null]	Swift Air, LLC d/b/a Eastern Air Lines d/b/a Eastern (2018 -)

SOOOO, NOW Are We Done?

Time to Refactor

Discussion About Benefits Realized

Enablement of safe refactoring and test-driven development

Questions?

Connect with me:

https://www.linkedin.com/in/donaldsawyer

Engage with Improving:

https://improving.com/contact