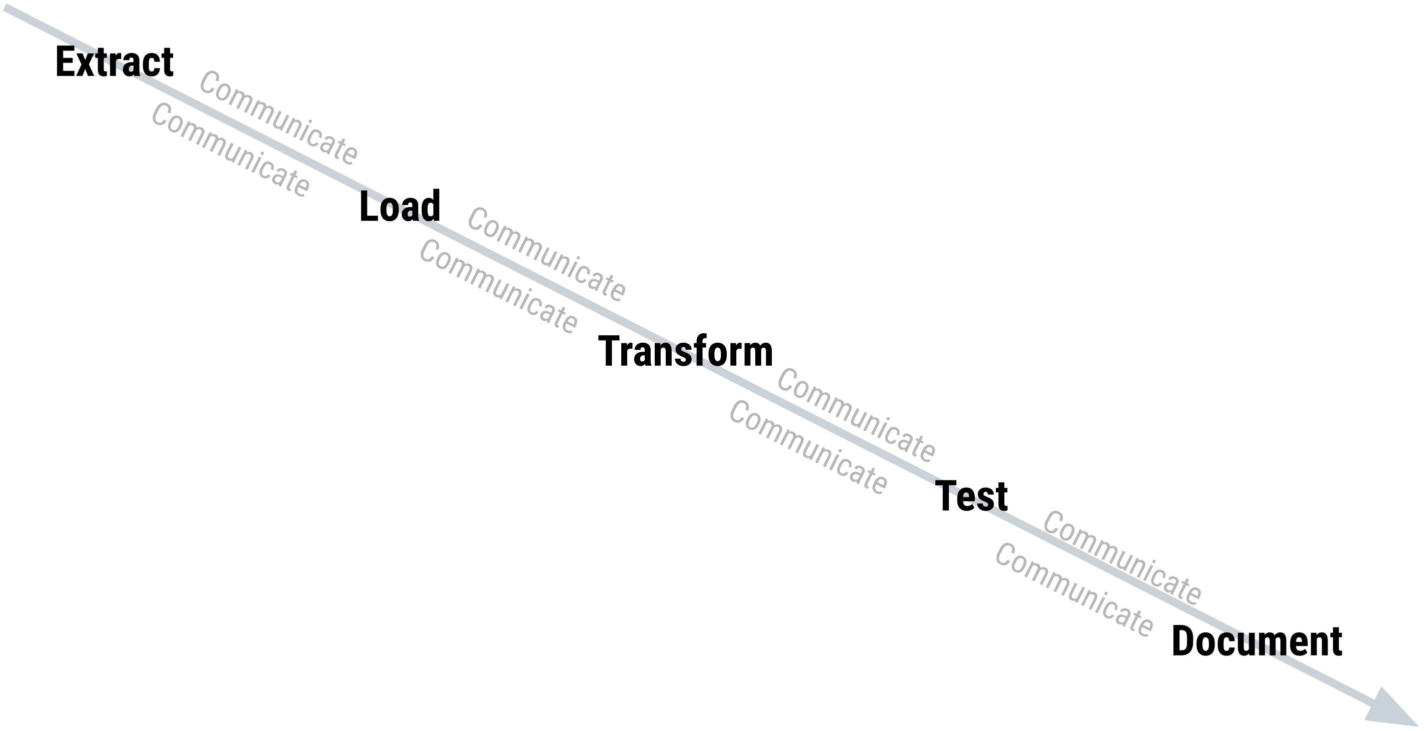
```
{“talk_title":
    “Operationalizing Column Name Contracts”,
 “talk_author”: {
   “author_name”: “Emily Riederer”,
   “author_twtr”: “@emilyriederer”,
   “author_site”: “emily.rbind.io”
  },
 “talk_forum”: {
   “forum_name”: “Coalesce”,
   “forum_locn”: “Online”,
   “forum_date”: “2021-12-07”
  }

}
```

Our tools solve the *technical* challenges but not *people* challenges



Extract

Communicate

Communicate

Load

Communicate

Communicate

Transform

Communicate

Communicate

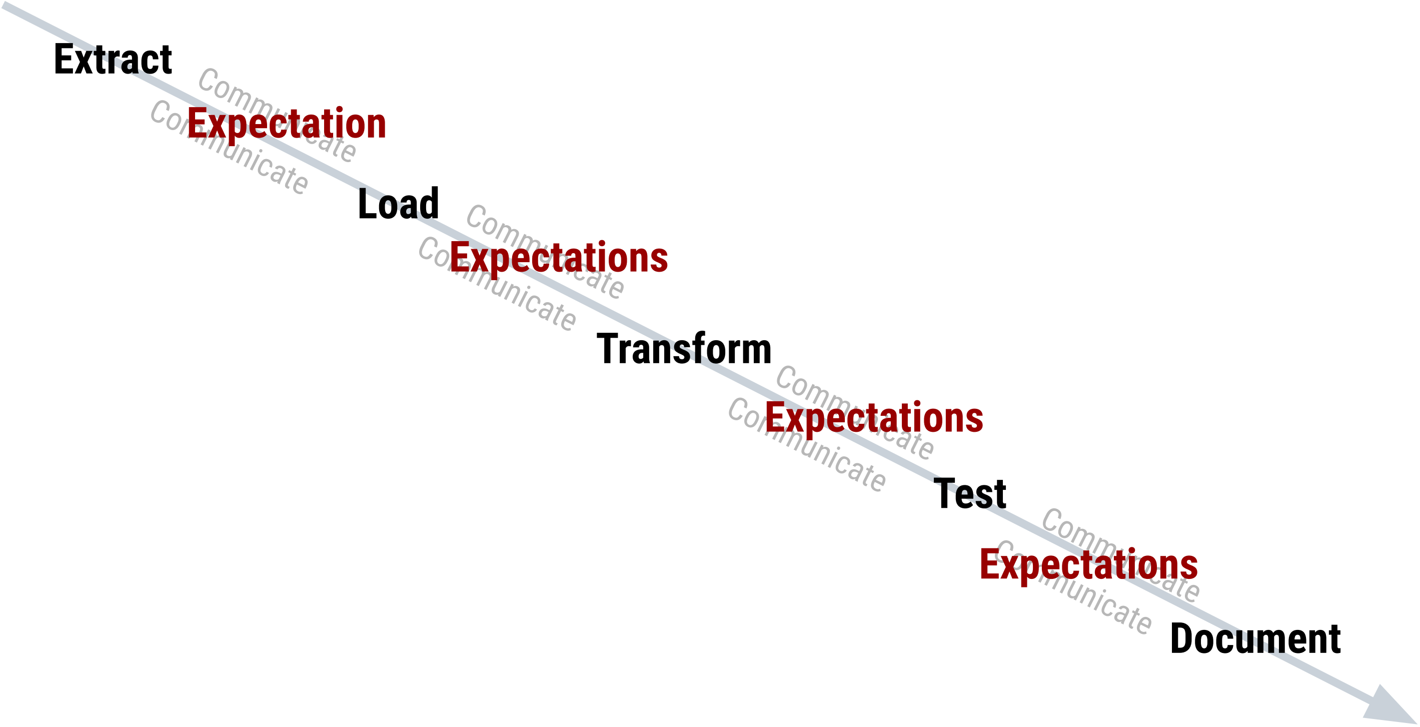
Test

Communicate

Communicate

Document

Technically-correct data is wrong if it isn’t fit to assumptions



Extract

Communicate

Expectation

Communicate

Load

Communicate

Expectations

Communicate

Transform

Communicate

Expectations

Communicate

Test

Communicate

Expectations

Communicate

Document

column
names
are contracts

column → interfaces

dev-to-user

names → configs

dev-to-dev

are... → code

dev-to-machine

column names are...

→ interfaces *dev-to-user*

→ configs *dev-to-dev*

→ code *dev-to-machine*

dbt + dbtplyr

column → interfaces
names → configs
are... → code

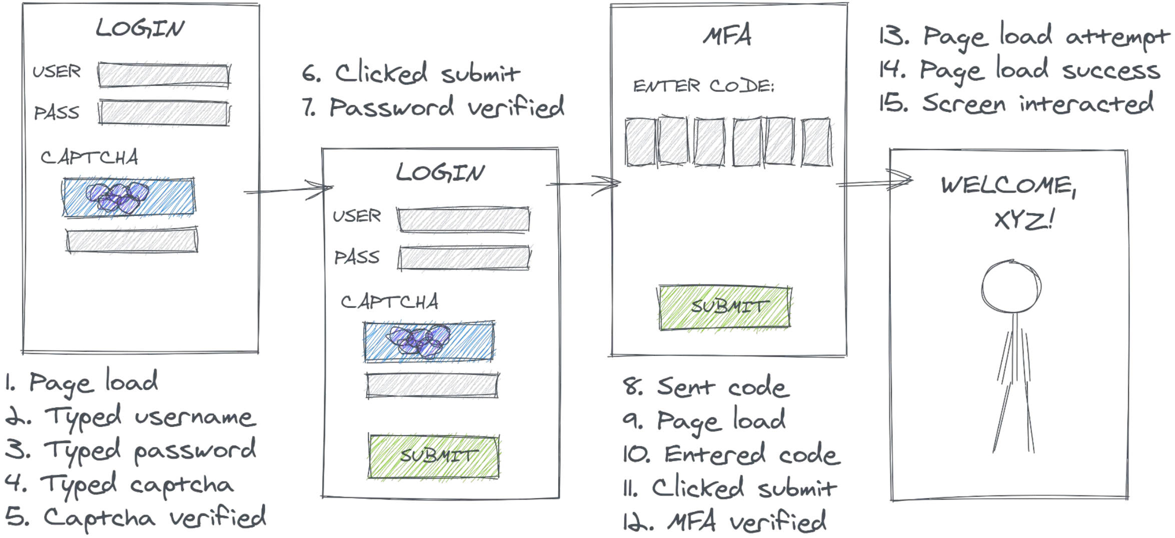
Column names are the user interface of our data

A	B	C	D
1	10	11	1
2	20	12	10
3	30	13	100
4	40	14	1,000
5	50	15	10,000
...	...	...	...

User Interface

Functionality

Data has functionality



1. Page load
2. Typed username
3. Typed password
4. Typed captcha
5. Captcha verified

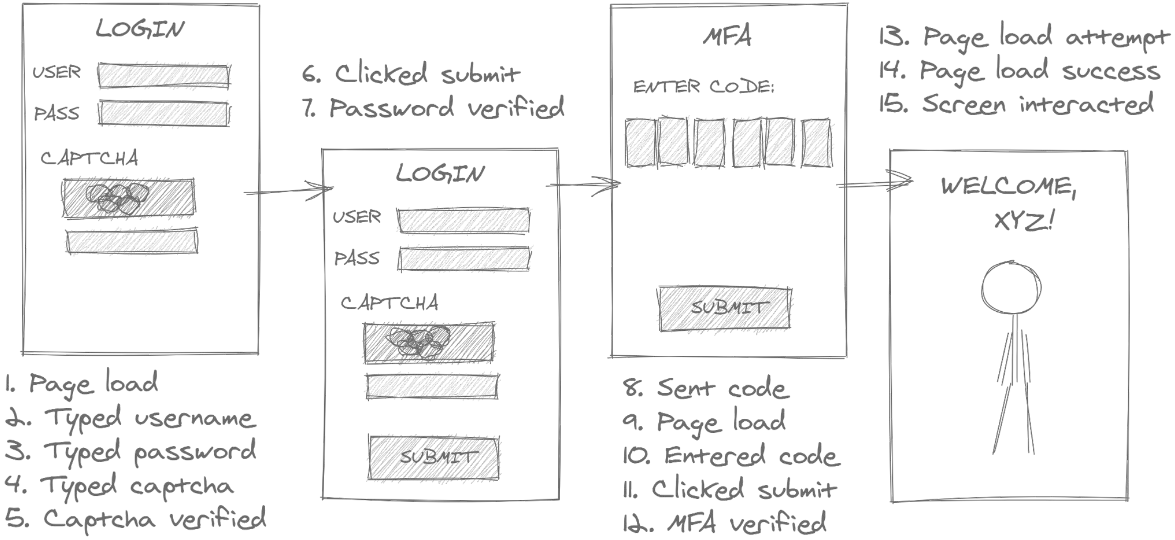
6. Clicked submit
7. Password verified

8. Sent code
9. Page load
10. Entered code
11. Clicked submit
12. MFA verified

13. Page load attempt
14. Page load success
15. Screen interacted

column names are… **interfaces** | configs | code

Data has functionality



1. Page load
2. Typed username
3. Typed password
4. Typed captcha
5. Captcha verified

6. Clicked submit
7. Password verified

8. Sent code
9. Page load
10. Entered code
11. Clicked submit
12. MFA verified

13. Page load attempt
14. Page load success
15. Screen interacted

Who is represented?

Where does it come from?

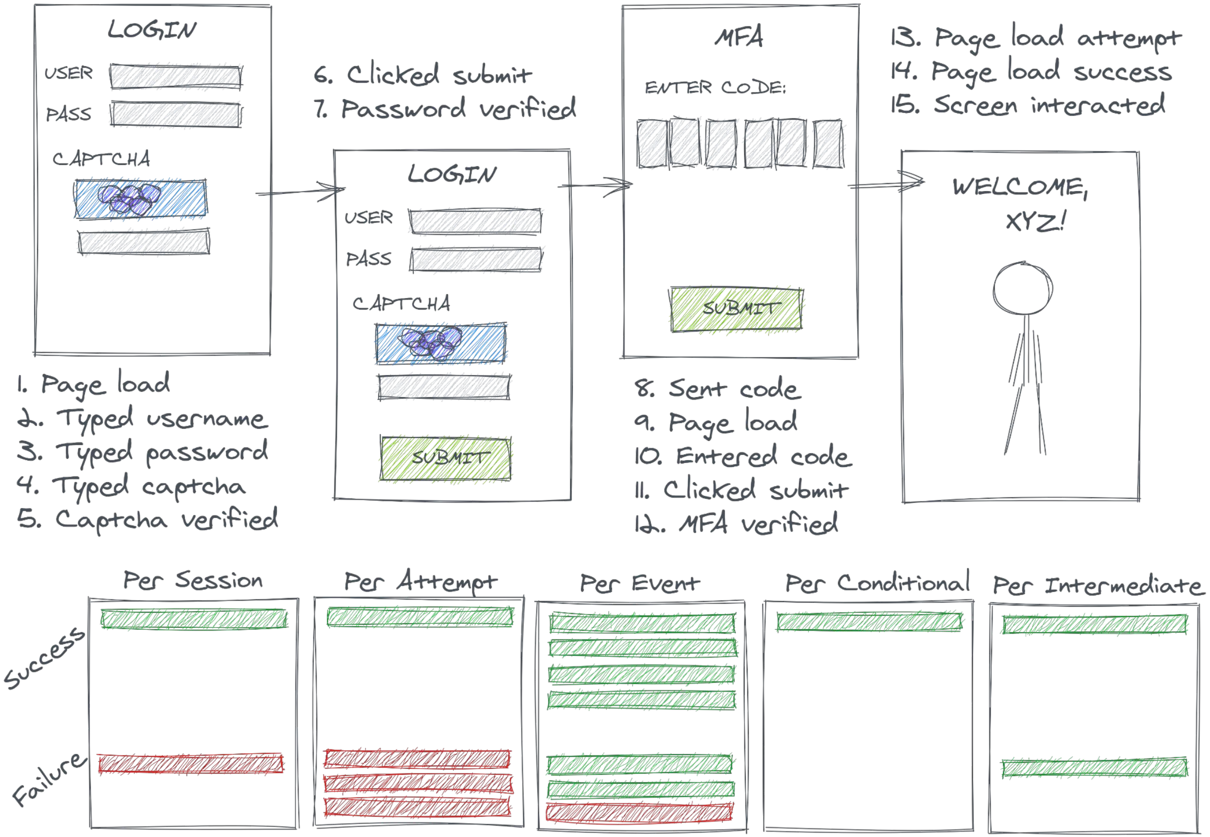
What are the keys?

How is it encoded?

When is it loaded?

column names are… **interfaces** | configs | code

Data has functionality



column names are… **interfaces** | configs | code

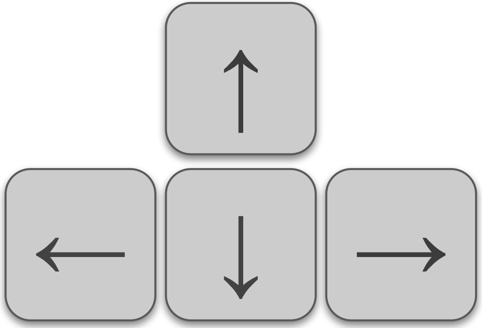
Interfaces make performance contracts

Universal Symbols

Intent

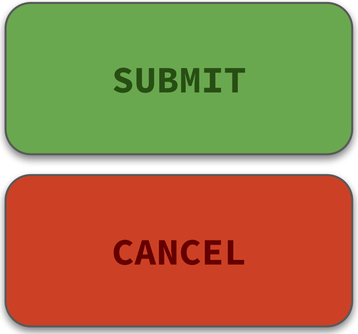
Grouping

Similarity



Aesthetics

Warning



column names are… **interfaces** | configs | code

Interfaces make performance contracts

Universal Symbols

Intent

“I am a binary variable”

Grouping

Similarity

“Here are all the binary variables in this dataset”

Aesthetics

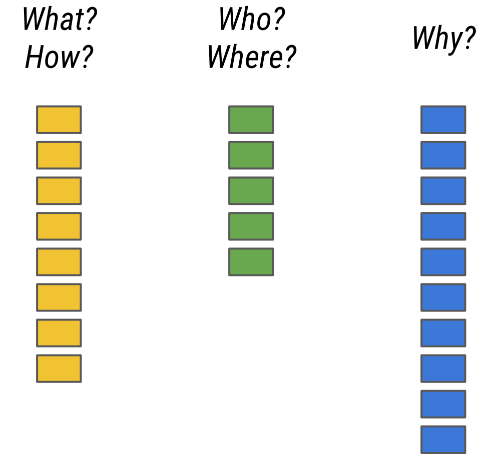
Warning

“Be careful - I may contain nulls”

Build a data interface with a controlled vocabulary

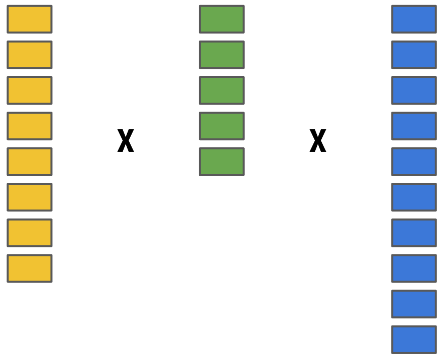
1. Define simple stubs

```
stub = semantics + contracts
```



2. Explain complex concepts

```
name = (type 1 stub)_(type 2 stub)_...
```



An example vocabulary

Stub
`ID`
`IND / IS`
`BIN`
`N`
`DT`
`...`

column names are… **interfaces** | configs | code

An example vocabulary

Stub	Semantics
ID	Unique entity identifier
IND / IS	Binary 0/1 indicator; rest of name describes 1 condition
BIN	Binary 0/1 indicator; rest of name describes 1 condition
N	Count of quantity or event occurrences
DT	Date of an event
...	…

An example vocabulary

Stub	**Semantics**
`ID`	Unique entity identifier
`IND / IS`	Binary 0/1 indicator; rest of name describes 1 condition
`BIN`	Binary 0/1 indicator; rest of name describes 1 condition
`N`	Count of quantity or event occurrences
`DT`	Date of an event
`...`	…

An example vocabulary

Stub	Semantics	Contracts
`ID`	Unique entity identifier	Numeric, primary / surrogate key
`IND / IS`	Binary 0/1 indicator; rest of name describes 1 condition	Always 0 or 1, non-null
`BIN`	Binary 0/1 indicator; rest of name describes 1 condition	Always 0 or 1
`N`	Count of quantity or event occurrences	Non-negative integer, non-null
`DT`	Date of an event	Date, ISO 8601 (YYYY-MM-DD)
`...`	...	...

An example vocabulary

Stub	Semantics	Contracts
`ID`	Unique entity identifier	Numeric, primary / surrogate key
`IND / IS`	Binary 0/1 indicator; rest of name describes 1 condition	Always 0 or 1, **non-null**
`BIN`	Binary 0/1 indicator; rest of name describes 1 condition	Always 0 or 1
`N`	Count of quantity or event occurrences	Non-negative integer, non-null
`DT`	Date of an event	Date, **ISO 8601** (YYYY-MM-DD)
`...`	...	...

An example vocabulary

Stub
`USER`
`LOGIN`
`...`

column names are… **interfaces** | configs | code

An example vocabulary

Stub
USER
LOGIN?
...

column names are… **interfaces** | configs | code

An example vocabulary

Stub	**Semantics**
`USER`	Unique site visitor as determined by IP address
`LOGIN`	A successful authentication (password, MFA) by a confirmed human actor (**after passing Captcha**)
`...`	…

column names are… **interfaces** \| configs \| code

An example vocabulary

Stub	Semantics	Consequence
`USER`	Unique site visitor as determined by IP address	Does **not unique**ly identify a person **across devices**
`LOGIN`	A successful authentication (password, MFA) by a confirmed human actor (**after passing Captcha**)	
`...`	…	…

column names are… **interfaces** \| configs \| code

An example vocabulary

Stub	**Semantics**	**Consequence**
`USER`	Unique site visitor as determined by IP address	Does **not unique**ly identify a person **across devices**
`LOGIN`	A successful authentication (password, MFA) by a confirmed human actor (**after passing Captcha**) ~~A session beginning with a visit to the login screen~~ ~~The click of the login button after typing username and password~~	
`...`	…	…

column names are… **interfaces** \| configs \| code

An example vocabulary

Types
ID
IND / IS
BIN
N
AMT
VAL
DT
TM
CAT
...

X

Subjects
USER
LOGIN
SESSION
CLICK
...

X

Details
UTM
DURATION
...

```
{DT | TM}_{LOGIN | SESSION}

ID_{USER | SESSION | LOGIN | VIEW}

{CAT | CD}_SOURCE_UTM

{CAT | CD}_MEDIUM_UTM

AMT_{SESSION | VIEW}_DURATION

...
```

column names are… **interfaces** | configs | code

Interfaces make performance contracts

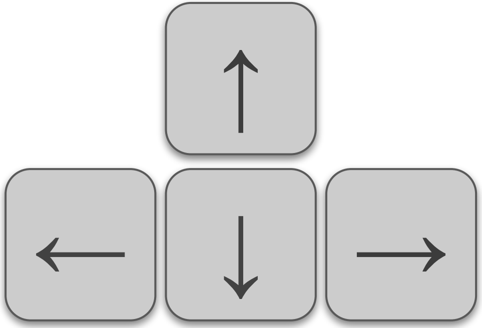
Universal Symbols

Intent

Programmatic wrangling

Grouping

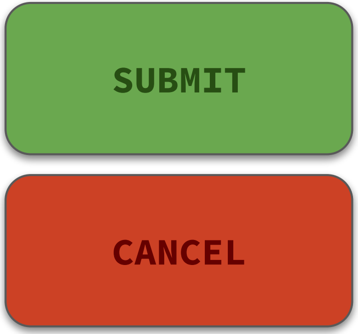
Similarity



Discoverability & documentation

Aesthetics

Warning



“Type hints”

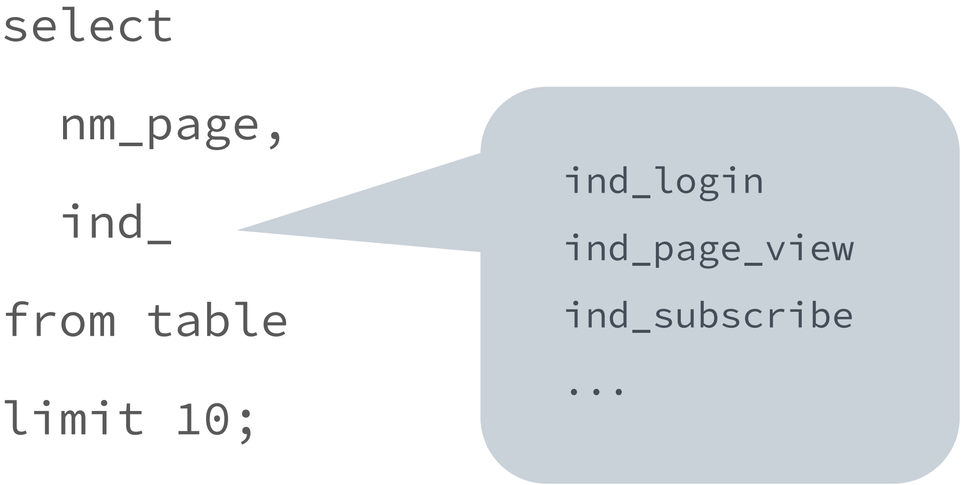
Universal symbols make it easier to wrangle the data

```
import pandas as pd

cols_ind = [vbl for vbl in data.columns if vbl[0:2] == 'IND_']

cols_grp = ["NM_PAGE"]

data.groupby(cols_grp)[cols_ind].mean()
```

```
#>                   IND_SUBSCRIBE
#>   NM_PAGE
#>  Version 1              0.149
#>  Version 2              0.235
#>  Version 3              0
```

column names are… **interfaces** | configs | code

Data UIs **group things** so it’s easier to find the data



```
select
  nm_page,
  ind_
from table
limit 10;
```

```
ind_login
ind_page_view
ind_subscribe
...
```

Data UIs **caution** users not to be deceived by the data

SUBMIT

CANCEL

Passed Captcha?	✗ LOGIN	✓ IND_LOGIN	✓ BIN_LOGIN
No	*NA*	0	*NA*
No	*NA*	0	*NA*
Yes	0	0	0
Yes	1	1	1
Yes	1	1	1

```
avg(LOGIN)= ⅔ -> P(LOGIN | CAPTCHA)

avg(IND_LOGIN)= ⅖ -> P(LOGIN)

avg(coalesce(BIN_LOGIN, 0)) = ⅖
```

column names are… **interfaces** | configs | code

column → interfaces

names → configs

are... → code

column → interfaces

names → configs

are... → code

Config files efficiently collect inputs

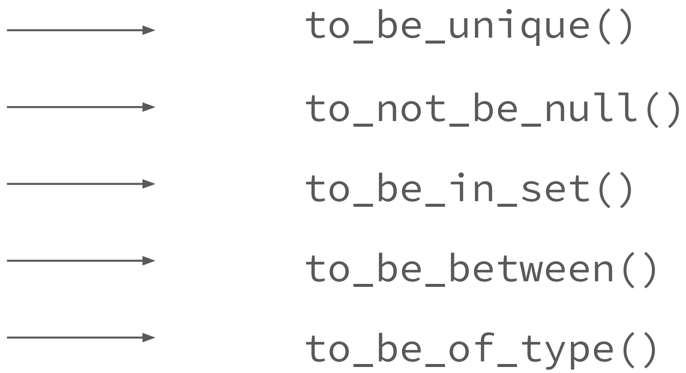
```yaml
name: 'dbtplyr'
version: '0.2.0'
config-version: 2
require-dbt-version: ">=0.19.0"

profile: 'dbtplyr'

source-paths: ["models"]
analysis-paths: ["analysis"]
test-paths: ["tests"]
data-paths: ["data"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]

target-path: "target"
clean-targets:
    - "target"
    - "dbt_modules"
```

Config files translate inputs to actions

great_expectations

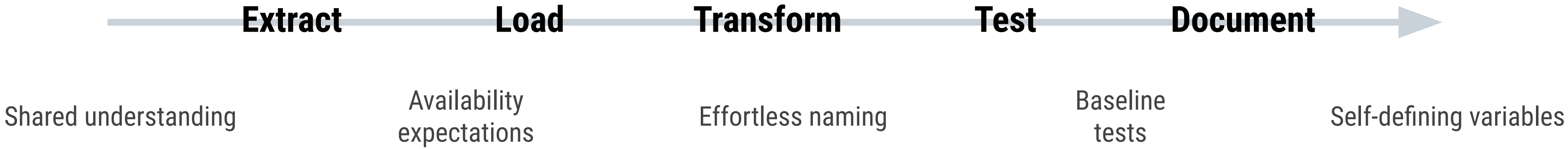
Stub	Contracts
`ID`	Numeric, primary / surrogate key
`IND / IS`	Always 0 or 1, non-null
`BIN`	Always 0 or 1
`N`	Non-negative integer, non-null
`DT`	Date, ISO 8601 (YYYY-MM-DD)
`...`	…

`expect_column_values_`



```
to_be_unique()
to_not_be_null()
to_be_in_set()
to_be_between()
to_be_of_type()
```

Config files are “input once, use everywhere”



Extract **Load** **Transform** **Test** **Document**

Shared understanding

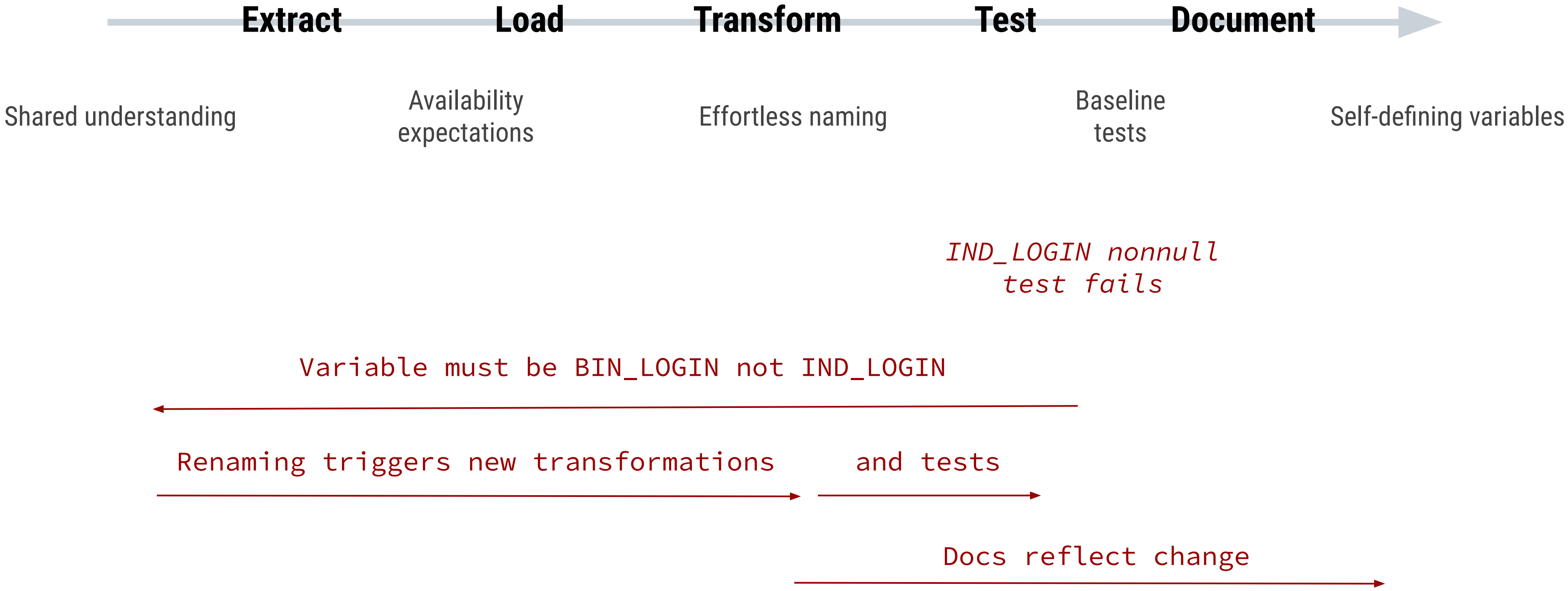
Availability
expectations

Effortless naming

Baseline
tests

Self-defining variables

Config files are “change once, update everywhere”



Extract **Load** **Transform** **Test** **Document**

Shared understanding

Availability expectations

Effortless naming

Baseline tests

Self-defining variables

IND_LOGIN nonnull test fails

Variable must be BIN_LOGIN not IND_LOGIN

Renaming triggers new transformations

and tests

Docs reflect change

column → interfaces

names → configs

are... → code

column → interfaces

names → configs

are... → code

Bad contracts are worse than no contracts

Inconsistency

Misspelled or free-style column names

Infidelity

Incorrect transformation based on contracts

Evasion

Creating problems instead of fixing

Code methodically turns theory to practice

Consistency

Jinja templates

Create valid names and avoid typos

Fidelity

Custom macros + dbtplyr

Iteratively apply transformation based on columns names

Accuracy

In-pipeline testing

Test validity of operations and contract adherence

dbtplyr helps maximize the benefits of column-name contracts

Key Functions	

Subset columns by name	`starts_with()`
	`ends_with()`
	`contains()`
	`not_contains()`
	`one_of()`
	`not_one_of()`
	`matches()`
	`everything()`

Iterate over transformations	`across()`
	`c_across()`

Iterate over filters	`if_any()`
	`if_all()`

inspired by R's dplyr syntax!

dbtplyr helps maximize the benefits of column-name contracts

Key Functions

Subset columns by name

```
{% set cols =
          dbtplyr.get_column_names(ref(‘data’)) %}
{% set cols_ind =
          dbtplyr.starts_with(cols, ‘ind’) %}
{% set cols_notnull = [‘x’, ‘y’] %}
```

```
[‘x’,‘y’,‘ind_a’,‘ind_b’]
```

Iterate over transformations

Iterate over filters

column names are… interfaces | configs | **code**

Broken contracts frustrate users

	✓	✓	✗	✗
ID_VARIANT	**N_CLICK_07**	**N_CLICK_14**	**N_CLIK_21**	**N_28_CLICK**
1	100	172	202	291
2	112	136	154	191
3	156	181	202	235

```
select
  n_click_07,
  n_clik_14..?
from table
```

column names are… interfaces | configs | **code**

Jinja templates enforce consistent naming and definitions

```
{% set lags = [‘07’,’14’,’21’,’24’]%}

select

  id_variant,

  {% for l in var('lags') %}

      count_if(n_days <= {{l}})
          as n_click_{{l}}

  {% if not loop.last %},{% endif %}
  {% endfor %}
```

```
select

  id_variant,

    count_if(n_days <= 07)
        as n_click_07,

    count_if(n_days <= 14)
        as n_click_14
```

column names are… interfaces | configs | **code**

Broken contracts lie to users

```
select count(*)

from logins

where dt_login = ‘2021-01-01’
```

DT_LOGIN	ID_LOGIN	IND_LOGIN
✗ 2021-01-01T 10:25:28	123	1
✗ 2021-01-01T 02:10:53	456	1
✗ 2021-01-02T 07:20:00	789	0

DT_LOGIN	ID_LOGIN	IND_LOGIN
✓ 2021-01-01	123	1
✓ 2021-01-01	456	1
✗ 2021-01-02	789	0

column names are… interfaces | configs | **code**

Custom macros + dbtplyr enforce contracts systemically

```
{% set cols =
         dbtplyr.get_column_names( ref(‘data’) )
%}
{% set cols_n =
         dbtplyr.starts_with(cols, ‘n’) %}
{% set cols_dt =
         dbtplyr.starts_with(cols, ‘dt’) %}
{% set cols_ind =
         dbtplyr.starts_with(cols, ‘ind’) %}

select

  {{ dbtplyr.across(cols_n,
                    “cast({var} as int)
                     as n_{var}”)}},
  {{ dbtplyr.across(cols_dt,
                    “date({var})
                     as dt_{var})”)}},
  {{ dbtplyr.across(cols_ind,
                    “coalesce({c}, 0)
                     as ind_{var}”) }}
```

```
select

  cast(n_a as int64) as n_a,
  cast(n_c as int64) as n_c,

  date(dt_b) as dt_b,
  date(dt_d) as dt_d,

  coalesce(ind_b,0) as ind_b,
  coalesce(ind_c,0) as ind_c
```

column names are… interfaces | configs | **code**

Custom macros + dbtplyr enforce contracts systemically

```
{% set cols =
         dbtplyr.get_column_names( ref(‘data’) )
%}
{% set cols_n =
         dbtplyr.starts_with(cols, ‘n’) %}
{% set cols_dt =
         dbtplyr.starts_with(cols, ‘dt’) %}
{% set cols_ind =
         dbtplyr.starts_with(cols, ‘ind’) %}

select

  {{ dbtplyr.across(cols_n,
                    “cast({var} as int)
                     as n_{var}”)}},
  {{ dbtplyr.across(cols_dt,
                    “date({var})
                     as dt_{var})”)}},
  {{ dbtplyr.across(cols_ind,
                    “coalesce({c}, 0)
                     as ind_{var}”) }}
```

```
select

  cast(n_a as int64) as n_a,
  cast(n_c as int64) as n_c,

  date(dt_b) as dt_b,
  date(dt_d) as dt_d,

  coalesce(ind_b,0) as ind_b,
  coalesce(ind_c,0) as ind_c
```

column names are… interfaces | configs | **code**

Broken contracts evade detection

```
{{ dbtplyr.across(cols_n, “cast({var} as int) as n_{var}”)}}
```

N_A	N_B
12.00	3.25
19.00	4.67
27.00	8.99

✓	✗
N_A	N_B
12	3
19	5
27	9

column names are… interfaces | configs | **code**

Testing confirms any non-enforceable contracts are upheld

```
{% set cols = get_column_names(ref('prep')) %}
{% set cols_n = starts_with(cols, 'n') %}

select *
from {{ ref('my_source') }}
where

 {%- for c in cols_n %}

  abs({{c}} - cast({{c}} as int64)) > 0.01 or

 {% endfor %}

 FALSE
```

```
with dbt__CTE__INTERNAL_test as (

select *
from `db`.`dbt_emily`.`my_source`
where

    abs(n_a - cast(n_a as int64)) > 0.01 or
    abs(n_b - cast(n_b as int64)) > 0.01 or
    abs(n_c - cast(n_c as int64)) > 0.01 or

   FALSE
)

select count(*) from dbt__CTE__INTERNAL_test
```

Consistent but deviant standards break users' trust

	X	X	X	X
ID_VARIANT	NUM_CLICK_07	NUM_CLICK_14	NUM_CLICK_21	NUM_CLICK_28
1	100	172	202	291
2	112	136	154	191
3	156	181	202	235

```
select
  n_click_07,     ✓
  n_click_14,     ✓
  n_click_21,     ✓
  n_click_28      ✓
from table
```

column names are… interfaces | configs | **code**

Test names - not just values

cols

COLUMN_NAME	L1	L2
IND_LOGIN	IND	LOGIN
PROP_LOGIN	PROP	LOGIN
NUM_LOGIN	NUM	LOGIN

Allowed Names

```
with cols as (

select
  column_name,
  split(lower(column_name), '_', 1) as l1,
  split(lower(column_name), '_', 2) as l2
from
  {{ ref(‘tbl’).database }}.
    {{ ref(‘tbl’).schema }}.
      INFORMATION_SCHEMA.COLUMNS
where table_name = '{{ ref(‘tbl’).identifier }}'

)
```

column names are… interfaces | configs | **code**

Test names - not just values

Data Types

```
with cols_type as (

select distinct
  split(lower(column_name), '_', 1) as stub,
  data_type
from
  {{ ref(‘tbl’).database }}.
    {{ ref(‘tbl’).schema }}.
      INFORMATION_SCHEMA.COLUMNS
where table_name = '{{ ref(‘tbl’).identifier }}'

)
```

cols_type

STUB	DATA_TYPE
N	INT64
PROP	FLOAT64
ID	INT64

column names are… interfaces | configs | **code**

Code methodically turns theory to practice

Consistency

Jinja templates

Create valid names and avoid typos

Fidelity

Custom macros + dbtplyr

Iteratively apply transformation based on columns names

Accuracy

In-pipeline testing

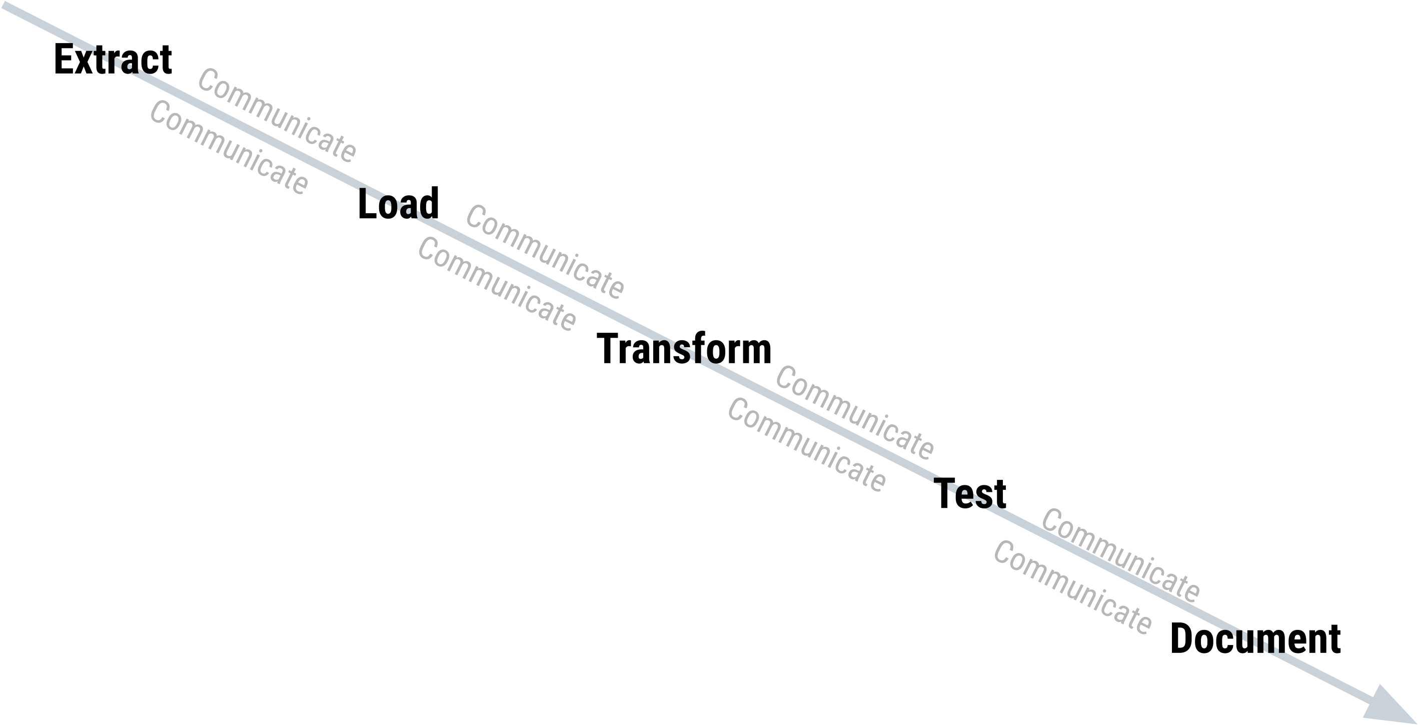
Test validity of operations and contract adherence

column → interfaces

names → configs

are... → code

Column names are contracts that persist through the data lifecycle



column names are… **interfaces | configs | code**

column
names
are contracts

```
{“talk_title":
    “Operationalizing Column Name Contracts”,
 “talk_author”: {
   “author_name”: “Emily Riederer”,
   “author_twtr”: “@emilyriederer”,
   “author_site”: “emily.rbind.io”
  },
 “talk_forum”: {
   “forum_name”: “Coalesce”,
   “forum_locn”: “Online”,
   “forum_date”: “2021-12-07”
  }

}
```