Reasoning Reliability in Wrike’s Data Pipeline
Wrike - A Collaborative Work Management Platform

Founded in 2006
10 Offices Globally
20,000+ Customers Globally
1000+ Employees
5 years in the Fast 500
20,000+

Organizations choose Wrike to orchestrate their digital work

With an additional 35,000 starting trials each month

- 2M users
- 130+ countries
- 10 languages
- 100M+ completed tasks
iOS and Android app icon for Beautyfilter

What needs to be designed? Icon
For what purpose it needs to be designed? Icon for mobile application - Beautyfilter
Do you have a reference idea on how it should look? Crown/Magic Stick/Mirror

Coordinator Anna
@Designer Matt Can you please assist with the design of the icon please?

Designer Matt
@Coordinator Anna Got the first draft here, let me know what you think.
### Web designers' workload

<table>
<thead>
<tr>
<th></th>
<th>Search by name</th>
<th>Oct 28 – Nov 3</th>
<th>Nov 4 – 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Su 28</td>
<td>Mo 29</td>
</tr>
<tr>
<td>Collin</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Kate</td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Xavier</td>
<td></td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

**Version comparison**

**Image for Summer Sale.jpg**

- **Version 4**
  - Collin 1 day ago
  - @Xavier, please remove rivets from panels.

- **Version 5**
  - Collin 5 mins ago
  - @Xavier, looks like you missed this.
Intro (0 out of 3)
Data Engineering in Wrike

- SaaS means that we
  - Create
  - Support
  - Sell our product, and
  - Attract leads
- Help these teams speak the language of data
- We’ve got big space for data democratization
Data Engineering Team in Wrike

- 16 data engineers in 4 teams
- We’re supporting 250+ DAGs on production
- Up to 1200 tasks
- With median of 13 tasks
- ~10 updates of production or acceptance each day
- Helped 5 other teams to start using Airflow
- ~10-15% of our colleagues are using data engineering infrastructure and sources every month directly (>50% are using analytical reports or through integrations)
We’ve Started With

● First analysts using new Data Warehouse based on Google BigQuery
● Data provided by a single instance of Airflow
  ○ A lot of bugs found on production data
  ○ A lot of changes during review
  ○ A lot of delays in data
  ○ Partially available data
  ○ Lack of the full picture during code review and architecture problems
● And we wanted to start democratization
  ○ Reliable production
  ○ No changes on production, at least unexpected ones
    ■ No changes in Data Structure
    ■ No changes in Data Freshness
## Acceptance Could Help

**DEV**
- Quickly run your pipeline on a very small subset of your data
- In our case 0.0025% of all data
- Nothing will make sense, but it’s a nice integration test

**TST**
- Select a subset of your data for data that you know
- Immediately see if something is off
- Still quick to run

**ACC**
- Carbon copy of production
- You can check if you feel comfortable pushing to PRD
- Give access to a Product Owner for them to check

**PRD**
- Greenlight procedure for merging from ACC to PRD
- Manual operation
- Great for git blame

*Via Data’s Inferno* by Wholesale Banking Advanced Analytics
Acceptance Environment

- Acceptance is an environment where changes are welcome
- To make sure that we aren’t going to need them on production
No Changes on Production, at Least Unexpected Ones

- No Changes in Data Structure
- No Changes in Data Freshness
- No Changes during release from Acceptance to Production
No Changes in Data Structure
# Implementation of Acceptance

## DEV
- Quickly run your pipeline on a very small subset of your data
- In our case 0.0025% of all data
- Nothing will make sense, but it’s a nice integration test

## TST
- Select a subset of your data for data that you know
- Immediately see if something is off
- Still quick to run

## ACC
- **Carbon copy of production**
- You can check if you feel comfortable pushing to PRD
- Give access to a Product Owner for them to check

## PRD
- Greenlight procedure for merging from ACC to PRD
- Manual operation
- Great for git blame

---

*No Changes in Data Structure (1 out of 3)*
Acceptance on DB Side. BigQuery

- Acceptance and production are **different projects** in the notation of BigQuery
- Isolated quotas and limits (resources)
- BigQuery allows for cross-project queries
  - So we store on acceptance only changed data
  - And take source data from production.
Dataflow Example

```
SELECT ...
FROM
`de-production.events.client`
GROUP BY ...
```

```
`de-acceptance.aggregations.client`
(v1)
```
No Changes in Data Structure (1 out of 3)

Dataflow Example

```
SELECT ...
FROM `de-production.events.client`
GROUP BY ...
```

```
`de-acceptance.aggregations.client` (v1)
```

```
`de-production.aggregations.client` (v1)
```
Dataflow Example

```
SELECT ...
FROM `de-production.events.client`
GROUP BY ...
```

```
`de-production.aggregations.client`
(v1)
```
Dataflow Example

SELECT ... 
FROM 
`de-production.events.client` 
GROUP BY ...

`de-acceptance.aggregations.client` (v2)

`de-production.aggregations.client` (v1)
Dataflow Example

```
SELECT ...
FROM `de-production.events.client`
GROUP BY ...
```

```
`de-acceptance.aggregations.client` (v2)
```

```
`de-production.aggregations.client` (v2)
```
Interface Separation on Other DBs

- Look for interface separation and resource isolation
  - And think about cost tradeoffs
- Approaches for **interface separation**
  - Schemas
  - Base directory name
  - Naming (bucket names for example)
  - Separate DBs
- Approaches for **resource isolation** (several trade offs with cost)
  - On service layer (separate DBs)
  - On DB side (e.g. roles, connection pools, quotas)
  - Airflow side (e.g. pools, priority, parallelism limit)
  - On monitoring side (e.g. query killer)
No Changes in Data Freshness
Beautiful DAG with 150 Tasks
Dataflow Example

```
SELECT ...
FROM `de-production.events.client`
GROUP BY ...

DAG: events loader (prod)
```

```
`de-acceptance.aggregations.client`
DAG: events aggregator (acc)
```

```
`de-production.aggregations.client`
DAG: events aggregator (prod)
```
Execution Example

DAG: events aggregator (acc)

DAG: events loader (prod)

DAG: events aggregator (prod)
Separate Airflows

- Coordinated via Postgres database named Partition Registry
  - Inspired by Functional Data Engineering by Maxime Beauchemin
  - Partition — unit of work for DAG, typically hour/day/week in a table
- State of partition published using operator
  - Explicitly publish sources
  - After all data validations have passed
- Wait for dependent sources using sensor
  - Automatically identify the strategy for interval
    - Week-on-hour, Month-on-day, custom catch-ups, etc.
Partition Registry Now

- Custom monitoring and alerts:
  - Severity of delays for partitions (DAG SLAs)
  - Base for data lineage
Partition Registry Now

- Custom monitoring and alerts:
  - Severity of delays for partitions (DAG SLAs)
  - Base for data lineage
- Not Airflow: Pentaho DI and Old Jenkins Pipelines
Partition Registry Now

- Custom monitoring and alerts:
  - Severity of delays for partitions (DAG SLAs)
  - Base for data lineage
- Not Airflow: Pentaho DI and Old Jenkins Pipelines
- Airflow for Analysts: isolated resources and credentials
Partition Registry Now

- Custom monitoring and alerts:
  - Severity of delays for partitions (DAG SLAs)
  - Base for data lineage
- Not Airflow: Pentaho DI and Old Jenkins Pipelines
- Airflow for Analysts: isolated resources and credentials
- K8s Airflow in Cloud
  - Easy switch with on-prem
  - Zero downtime migration
  - Data locality

No Changes in Data Freshness (2 out of 3)
No Changes During Release from Acc to Prod
Acceptance Told Us Where We Went Wrong

- 696 Commits
- 47 Branches

production airflow-d:

* acceptance
  - ed4b4018 · remove test_source from alerts · 2 days ago

* production
  - ff1c9375 · new non_pii_backend views · 2 days ago

No Changes During Release Process (3 out of 3)
Fast and Reliable Release

- We need code freeze to test dependent parts
- But we need 10 releases per day
  - So, we need to freeze as little as possible
    - But still review and test every change made
Dependency Scheme

DAG: saas_x

DAG: saas_y

DAG: x_aggregator

DAG: events_loader
Dependency Scheme with Code

- **DAG: events_loader**
- **DAG: x_aggregator**
- **DAG: saas_x**
- **DAG: saas_y**
- Common Operators
- Some other shared code
- Shared code for SAASes
No Changes During Release Process Means

- Good data isolation during release
- Good code isolation during release
Bad Data Isolation Is When

- You recalculate your data and get different results
- Data distribution changes
- Data distribution does not change when it should
- Analytical dashboard starts to focus on the wrong things
- You achieve your results a lot faster :)
- Something else is wrong and you don't know about it.
So if Data Changes

- It’s safe to assume
  - Review is no longer valid
  - Manual testing is no longer valid
  - Data sources may be corrupted

- So before the release of data change
  - Notifying all stakeholders of all changed dependent sources
  - Checking that everything works correctly on acceptance
  - Making atomic release

- We’re helping to implement recalculation strategies
  - Recalculating everything and keeping it up-to-date
  - Preserving history for metrics in prestaging
  - Supporting and gradual deprecation of old version of metrics
Keeping Track of Data Isolation

- Knowing when dependencies are updated after release to production
  - Notifications from other teams
  - Dependency on exact version of partition
    - Makes it easier to switch between acc and prod in code
  - Validation of data on your side
    - Great Expectations to explicitly specify your assumptions on data nature
    - Anomaly detection

- Finding all dependent sources before release to the production
  - Manual
    - BigQuery history
    - Search in git repository
  - Data Lineage + release process
  - Autotests
Good Code Isolation

- Bad code isolation means you have a bug and your pipeline is not working
- This happens when 2+ DAGs use the same code
  - You update code or library and other DAG fails
- Two types of failure
  - Scheduler/Web Server — appears immediately, hard isolation (fat-zip, boilerplate)
  - Worker — visible during execution, easy isolation (k8s, venv)
    - Can be at the end of a 4 hour-long task at the start of the next month :(
- How do we avoid this?
  - There is 20% of code used in 80% of cases
    - We’re moving it to the library, test and track backward compatibility
  - We have a shared code that is changed rarely
    - This code should be as private as possible to make sure that we’re not reusing it
    - The main reason for DAGs to be included in the single repo or merge request
Dependency Scheme with Code

- DAG: saas_x
- DAG: saas_y
- DAG: events_loader
- DAG: x_aggregator
- Some other shared code
- Shared code for SAASes

No Changes During Release Process (3 out of 3)
How Do We Reason About Reliability?

- Our production is very predictable
- All interface changes reviewed on separate environment
  - We keep track of all data dependencies and communicate the change to all stakeholders throughout the pipeline
  - Every source on production is reviewed, supported by several data engineers, have a clear time of readiness and all errors are communicated to all stakeholders
- We’re using partition registry
  - To isolate resources of acceptance
    - As little recalculation as possible
  - To integrate Airflow with separate creds and resources to other teams
- Acceptance could be made cheaper
Thank You!
Any Questions?

Alexander Eliseev at Airflow Slack
alexander.eliseev@team.wrike.com
https://github.com/eliseealex