Test strategies for data processing pipelines

Lars Albertsson, independent consultant (Mapflat)
Øyvind Løkling, Schibsted Products & Technology

Who’s talking?

Swedish Institute of Computer. Science. (test & debug tools)

Sun Microsystems (large machine verification)

Google (Hangouts, productivity)

Recorded Future (NLP startup) (data integrations)

Cinnober Financial Tech. (trading systems)

Spotify (data processing & modelling, productivity)

Schibsted Products & Tech (data processing & modelling)

Mapflat (independent data engineering consultant)

Agenda

- Data applications from a test perspective
- Testing stream processing product
- Testing batch processing products
- Data quality testing

Main focus is functional, regression testing

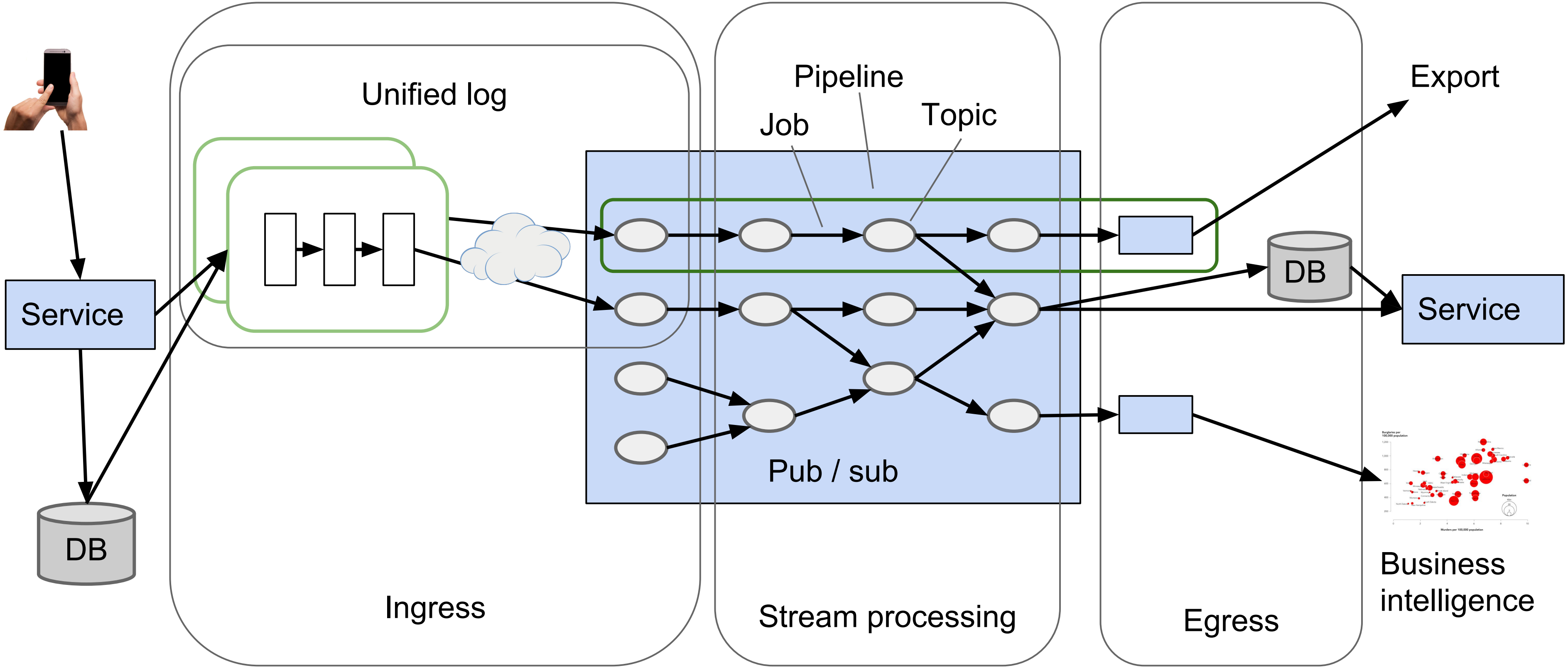
Prerequisites: Backend dev testing, basic data experience

Test value

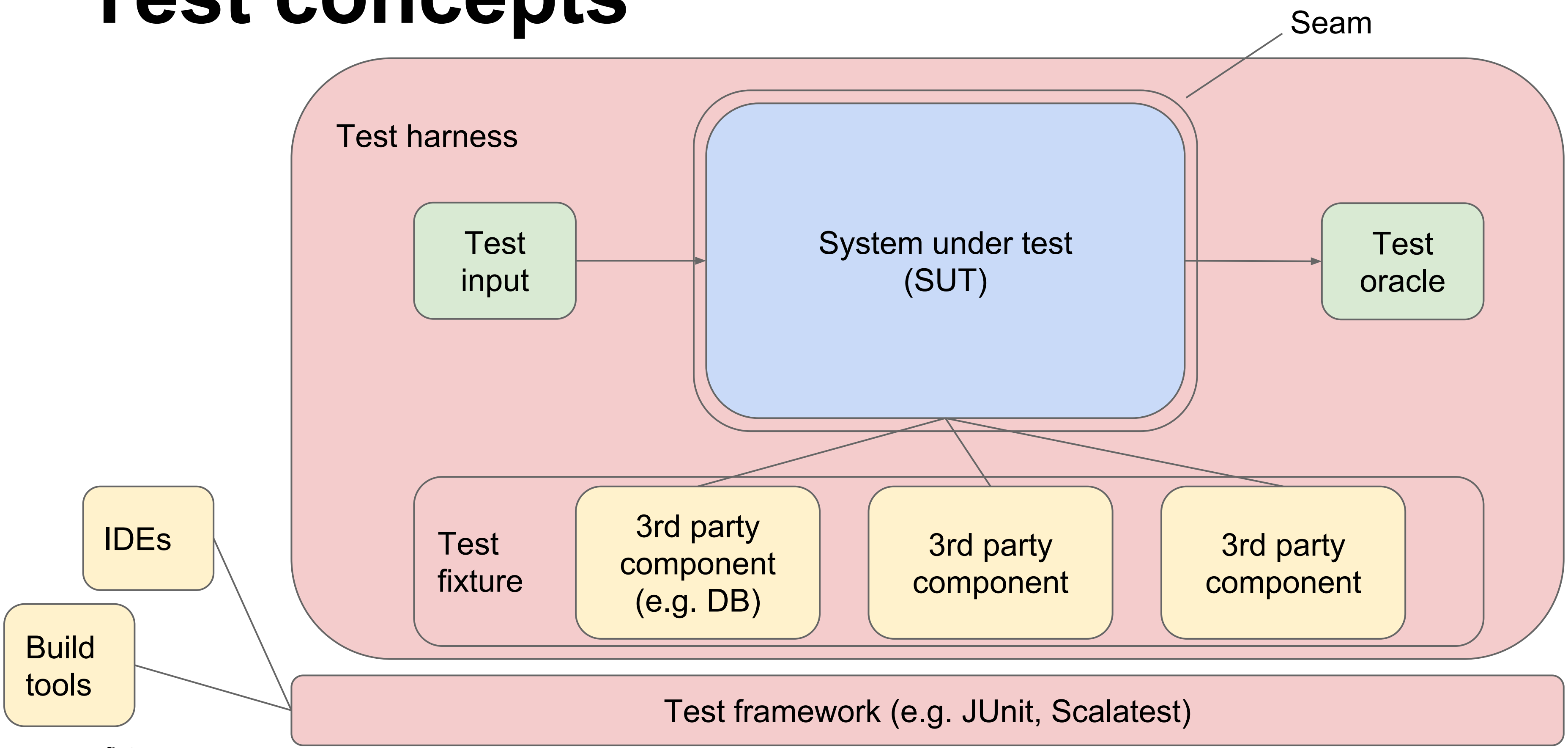
For data-centric applications, in this order:

- Productivity
  - Move fast without breaking things
- Fast experimentation
  - 10% good ideas, 90% bad
- Data quality
  - Challenging, more important than
- Technical quality
  - Technical failure => ops hassle, stale data

Streaming data product anatomy

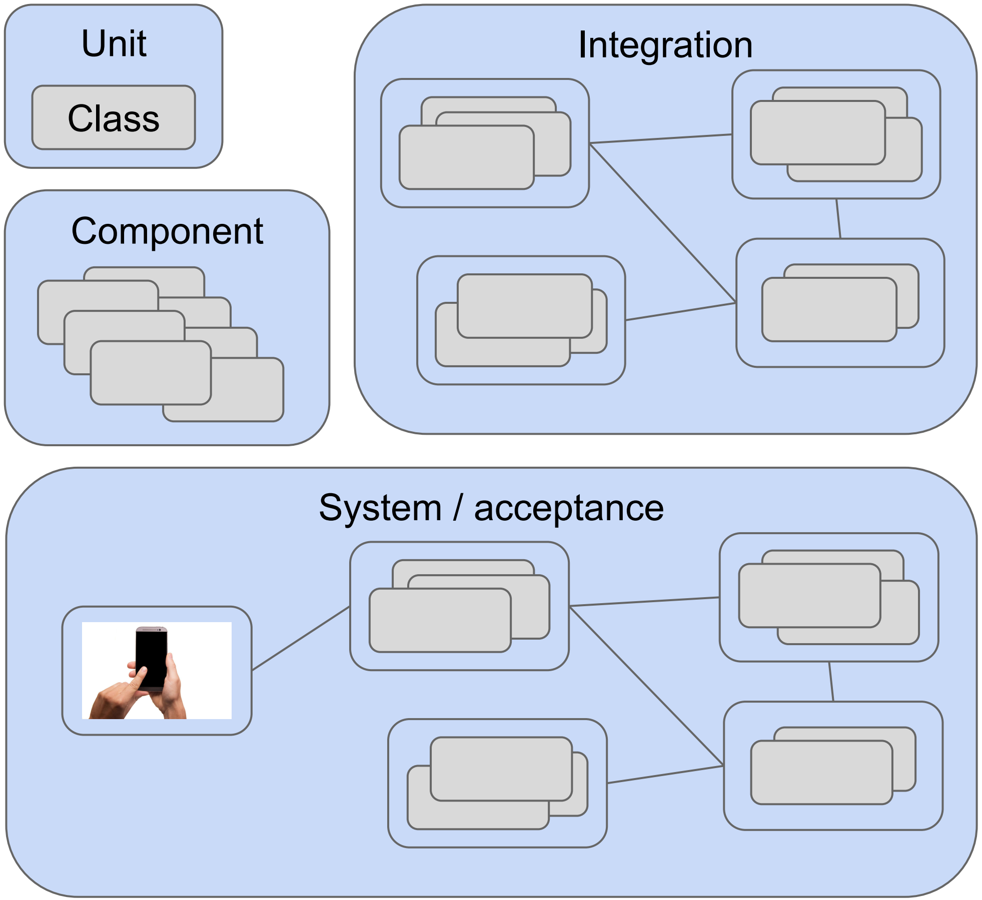


Test concepts



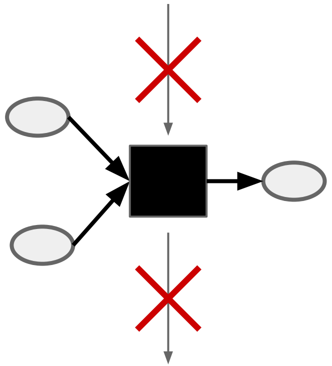
Test scopes

- Pick stable seam
- Small scope
  - Fast?
  - Easy, simple?
- Large scope
  - Real app value?
  - Slow, unstable?
- Maintenance, cost
  - Pick few SUTs

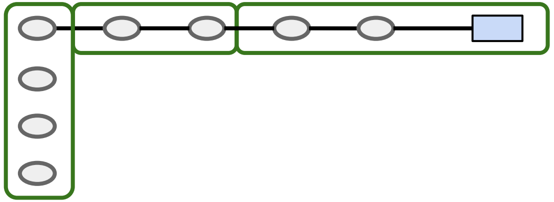


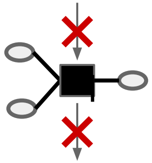
Data-centric application properties

- Output = function(input, code)
  - No external factors => deterministic
- Pipeline and job endpoints are stable
  - Correspond to business value
- Internal abstractions are volatile
  - Reslicing in different dimensions is common



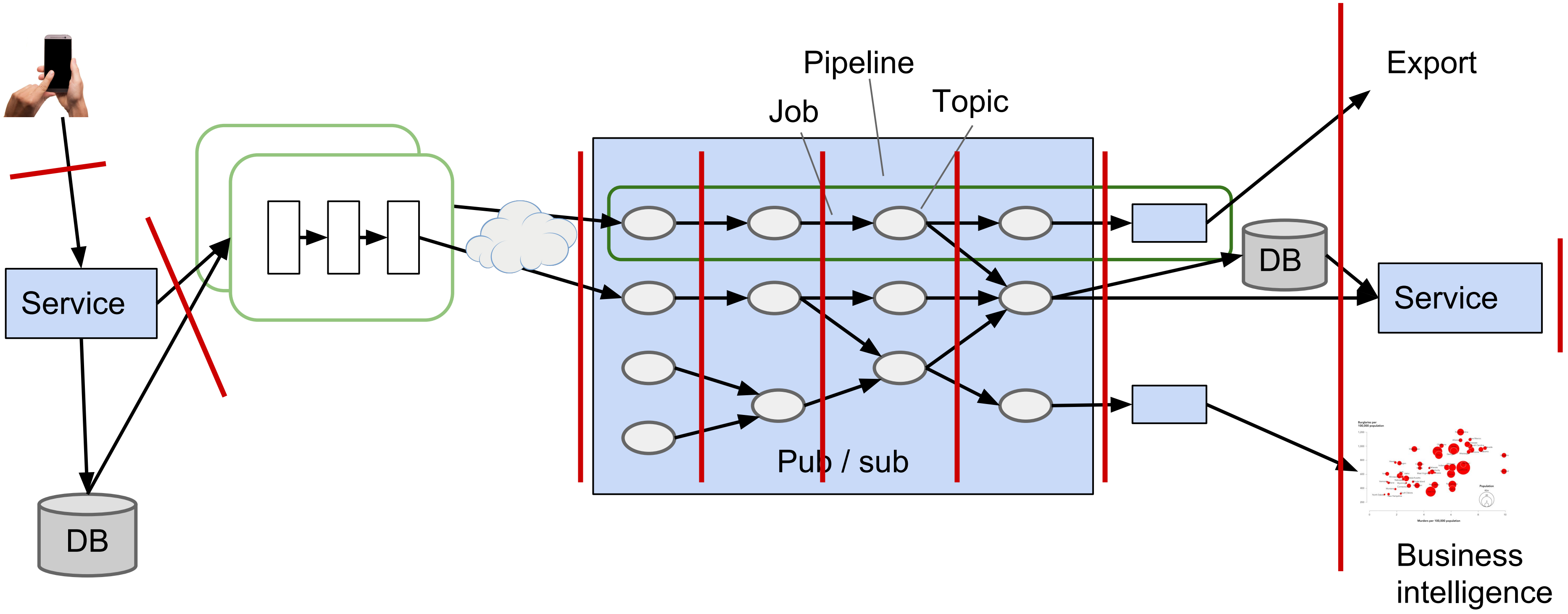
Data-centric app test properties



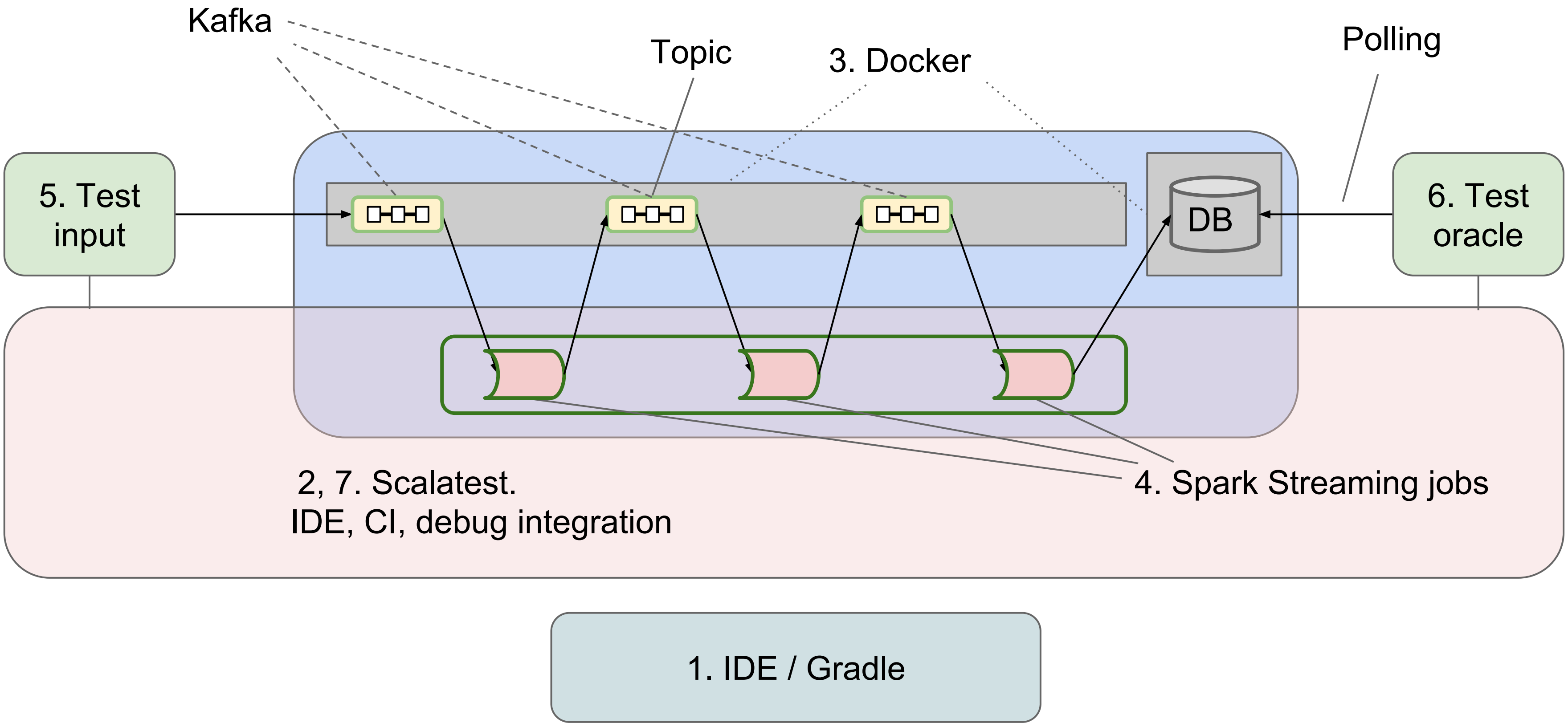


- Output = function(input, code)
  - **Perfect for test!**
  - Avoid: external service calls, wall clock
- Pipeline/job edges are suitable seams
  - Focus on large tests
- Internal seams => high maintenance, low value
  - *Omit unit tests, mocks, dependency injection!*
- Long pipelines crosses teams
  - Need for end-to-end tests, but culture challenge

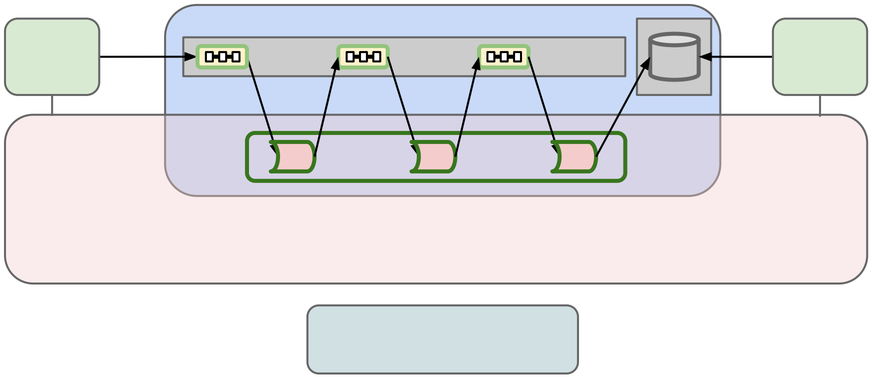
Suitable seams, streaming



Streaming SUT, example harness



Test lifecycle



1. Start fixture containers
2. Await fixture ready
3. Allocate test case resources
4. Start jobs
5. Push input data to Kafka
6. While (!done && !timeout) { pollDatabase(); sleep(1ms) }
7. While (moreTests) { Goto 3 }
8. Tear down fixture

For absence test, send dummy sync messages at end.

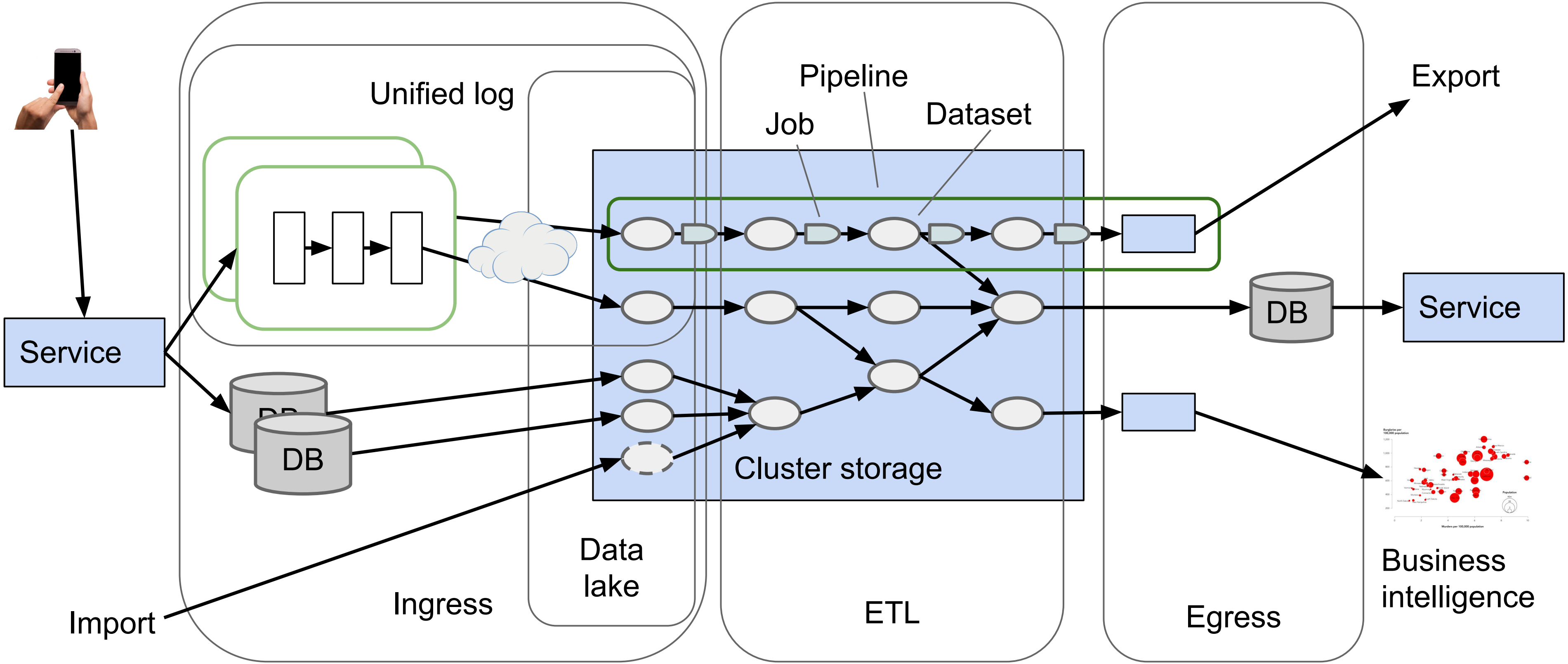
Input generation

- Input & output is denormalised & wide
- Fields are frequently changed
  - Additions are compatible
  - Modifications are incompatible => new, similar data type
- ~~Static test input, e.g. JSON files~~
  - Unmaintainable
- Input generation routines
  - Robust to changes, reusable

Test oracles

- ~~Compare with expected output~~
- Check fields relevant for test
  - Robust to field changes
  - Reusable for new, similar types
- Tip: Use lenses
  - JSON: JsonPath (Java), Play JSON (Scala)
  - Case classes: Monocle
- Express invariants for each data type

Batch data product anatomy



Datasets

- Pipeline equivalent of objects
- Dataset class == homogeneous records, open-ended
  - Compatible schema
  - E.g. MobileAdImpressions
- Dataset instance = dataset class + parameters
  - Immutable
  - Finite set of homogeneous records
  - E.g. MobileAdImpressions(hour=”2016-02-06T13”)

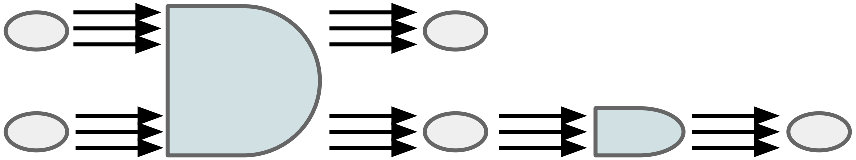
Directory datasets

Privacy level | Dataset class | Schema version | Instance parameters, Hive convention | Seal | Partitions

```
hdfs://red/pageviews/v1/country=se/year=2015/month=11/day=4/_SUCCESS
                                                            part-00000.json
                                                            part-00001.json
```

- Some tools, e.g. Spark, understand Hive name conventions

Batch processing



- outDatasets = code(inDatasets)
- Component that scale up
  - **Spark**, (Flink, Scalding, Crunch)
- And scale down
  - Local mode
  - Most jobs fit in one machine

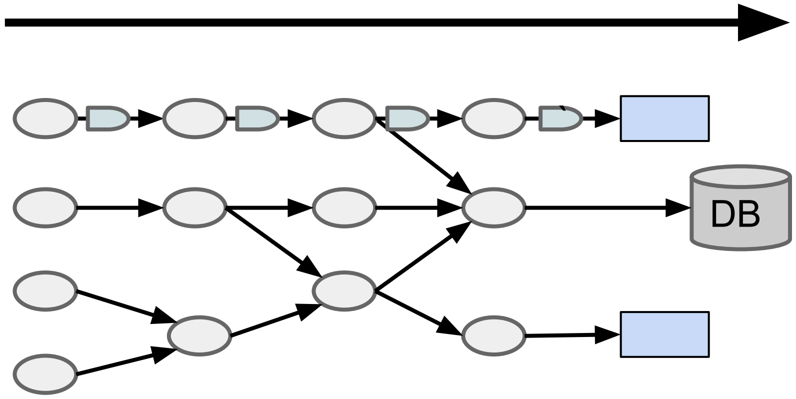
Gradual refinement

1. Wash
   - time shuffle, dedup, ...
2. Decorate
   - geo, demographic, ...
3. Domain model
   - similarity, clusters, ...
4. Application model
   - Recommendations, ...

Workflow manager

- Dataset "build tool"
- Run job instance when
  - input is available
  - output missing
- Backfills for previous failures
- DSL describes dependencies
- Includes ingress & egress

Suggested: Luigi / Airflow

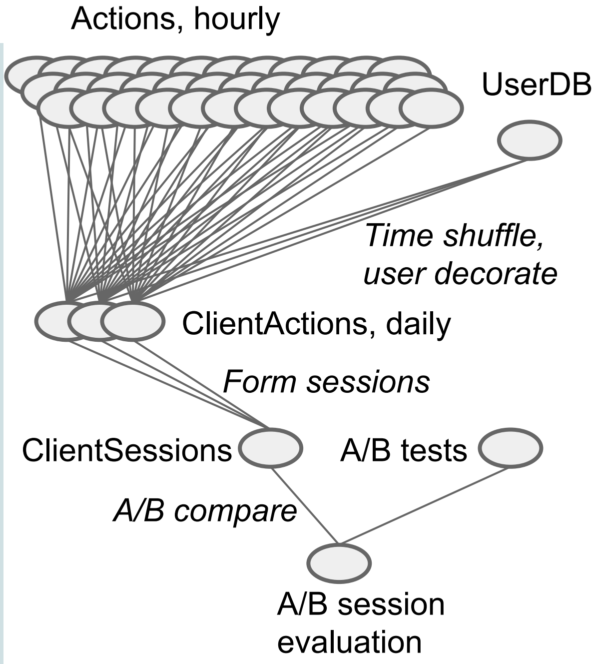


DSL DAG example (Luigi)

```
class ClientActions(SparkSubmitTask):
  hour = DateHourParameter()
  def requires(self):
    return [Actions(hour=self.hour - timedelta(hours=h)) for h in range(0, 24)] + \
      [UserDB(date=self.hour.date)]
  ...

class ClientSessions(SparkSubmitTask):
  hour = DateHourParameter()
  def requires(self):
    return [ClientActions(hour=self.hour - timedelta(hours=h)) for h in range(0, 3)]
  ...

class SessionsABResults(SparkSubmitTask):
  hour = DateHourParameter()
  def requires(self):
    return [ClientSessions(hour=self.hour), ABExperiments(hour=self.hour)]

  def output(self):
    return HdfsTarget("hdfs://production/red/ab_sessions/v1/" +
      "{:year=%Y/month=%m/day=%d/hour=%H}".format(self.hour))

  ...
```

Job (aka Task) classes

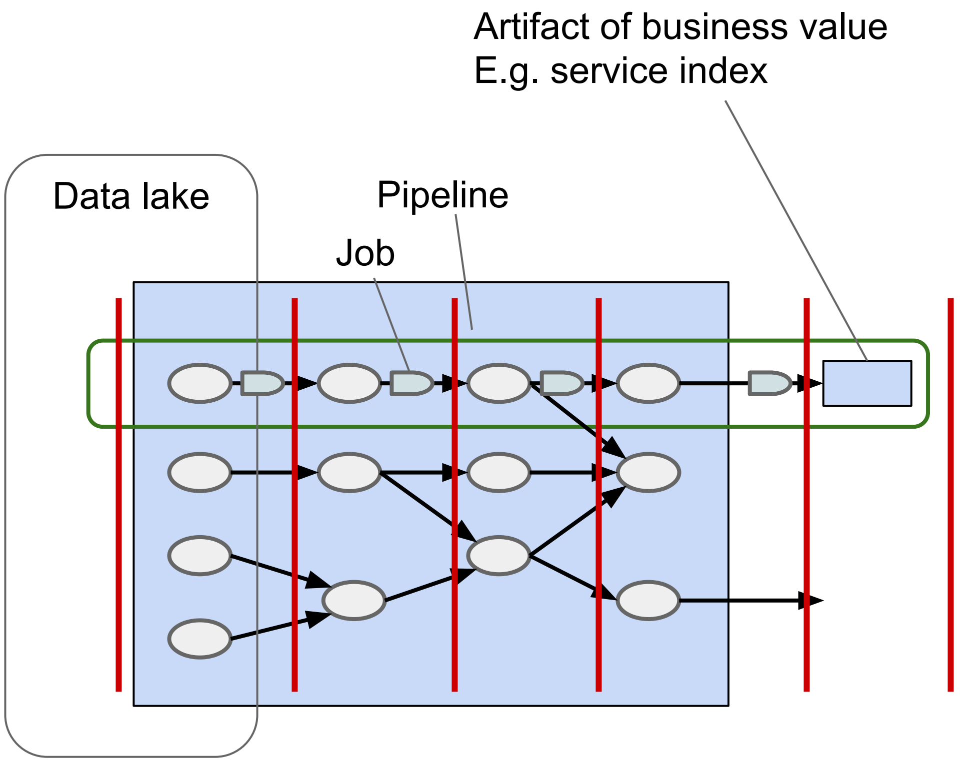
Dataset instance



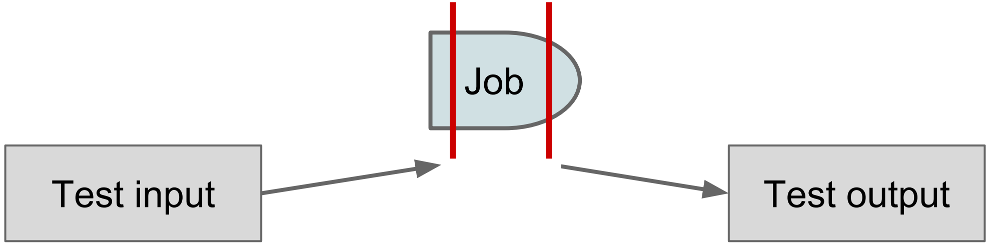
- Expressive, embedded DSL - a must for ingress, egress
  - Avoid weak DSL tools: ~~Oozie, AWS Data Pipeline~~

Batch processing testing

- Omit collection in SUT
  - Technically different
- Avoid clusters
  - Slow tests
- Seams
  - Between jobs
  - End of pipelines

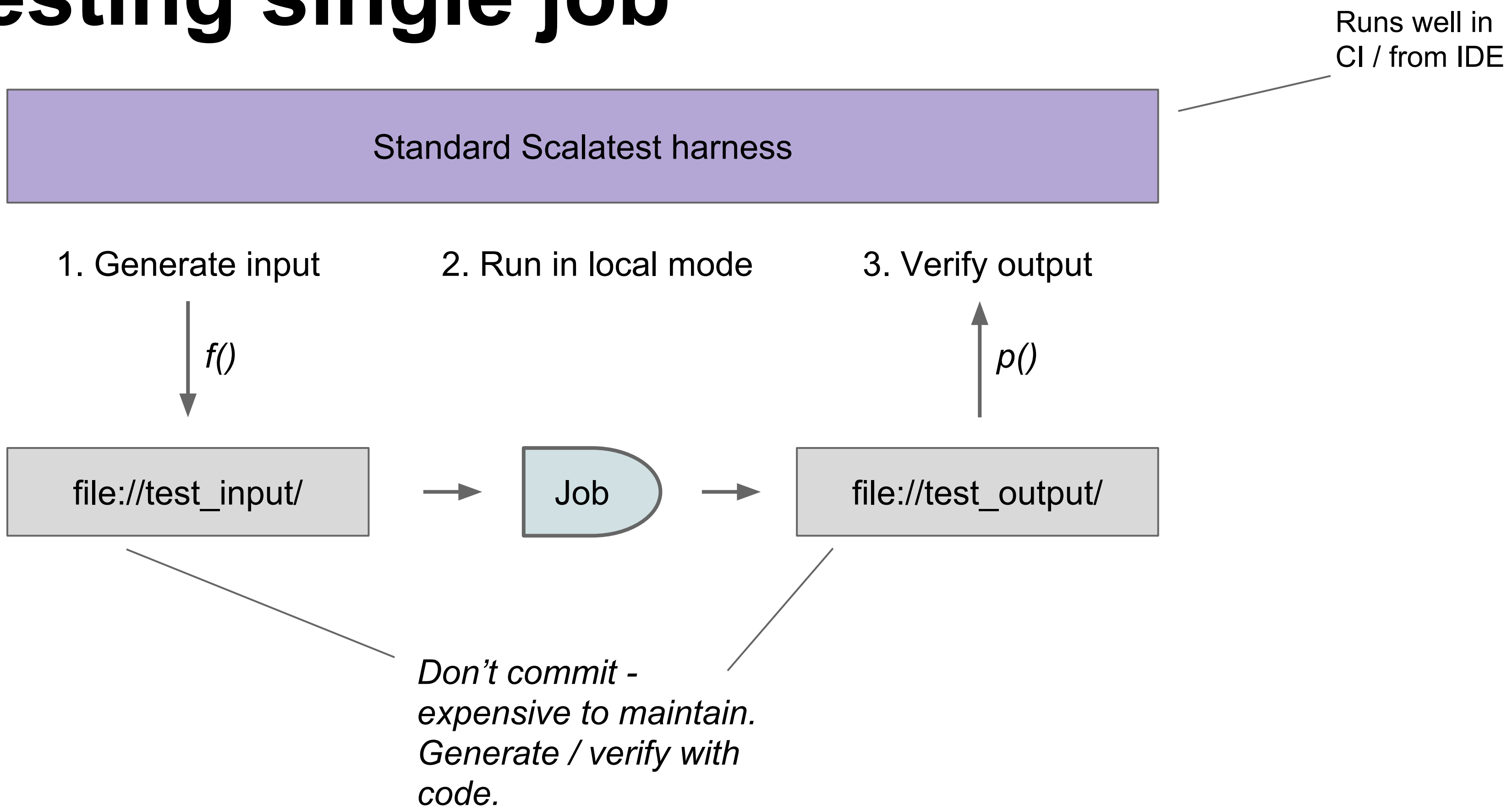


Batch test frameworks - don’t

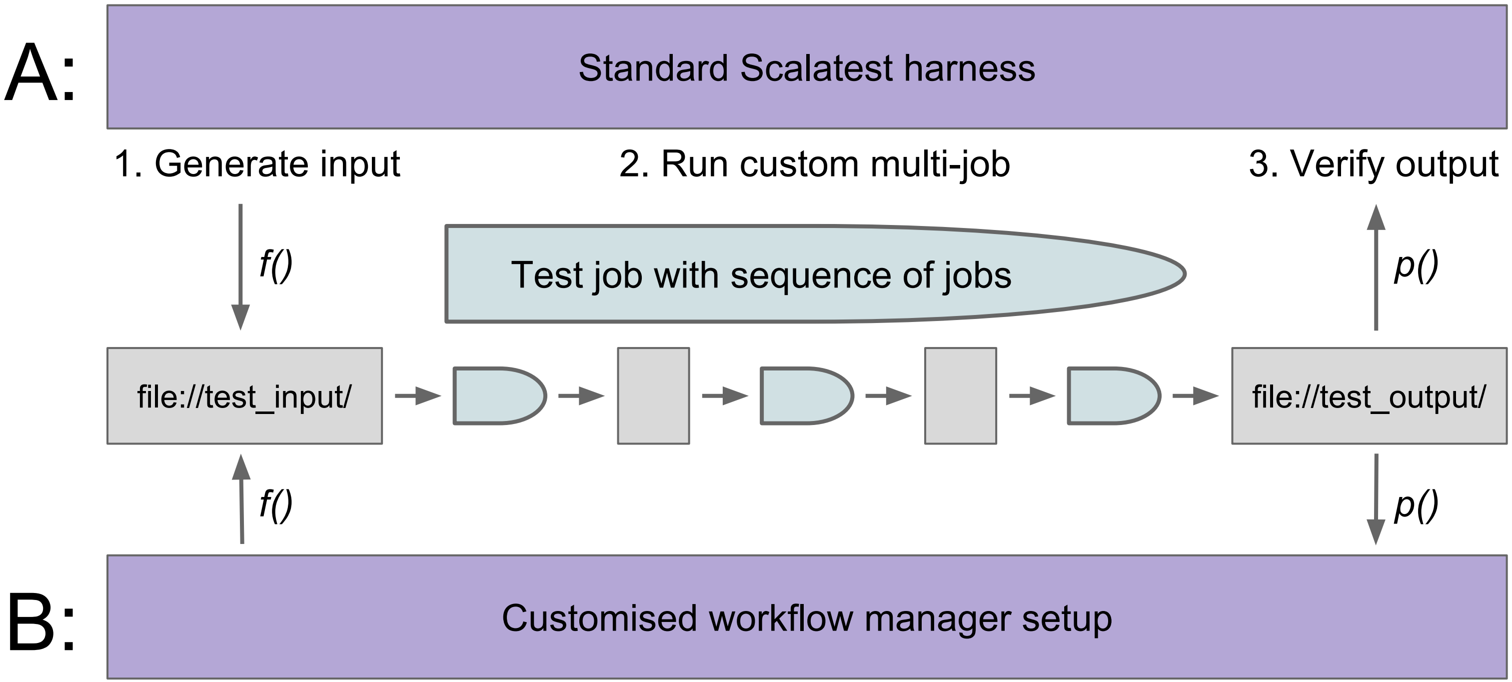


- Spark, Scalding, Crunch variants
- Seam == internal data structure
  - Omits I/O - common bug source
- **Vendor lock-in**
  When switching batch framework:
  - Need tests for protection
  - Test rewrite is unnecessary burden

Testing single job



Testing pipelines - two options



+ Runs in CI
+ Runs in IDE
+ Quick setup
- Multi-job maintenance

+ Tests workflow logic
+ More authentic
- Workflow mgr setup for testability
- Difficult to debug
- Dataset handling with Python

● Both can be extended with egress DBs

Testing with cloud services

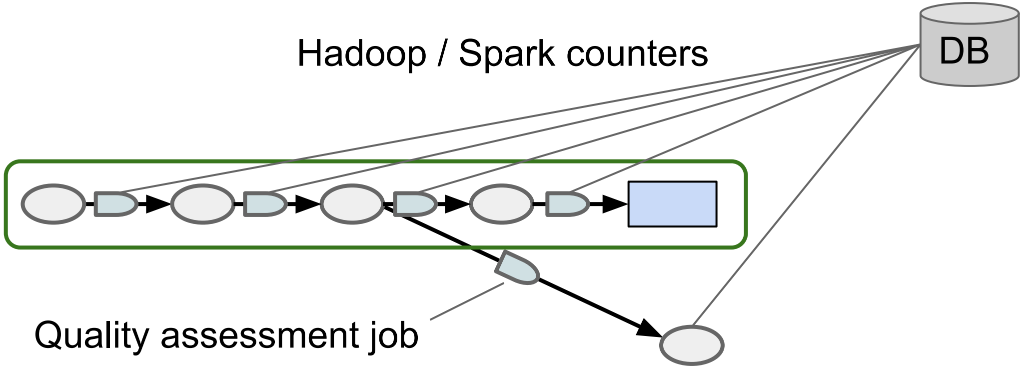
- PaaS components do not work locally
  - Cloud providers should provide fake implementations
  - Exceptions: Kubernetes, Cloud SQL, (S3)
- Integrate PaaS service as fixture component
  - Distribute access tokens, etc
  - Pay $ or $$$

Quality testing variants

- Functional regression
  - Binary, key to productivity
- Golden set
  - Extreme inputs => obvious output
  - No regressions tolerated
- (Saved) production data input
  - Individual regressions ok
  - Weighted sum must not decline
  - Beware of privacy

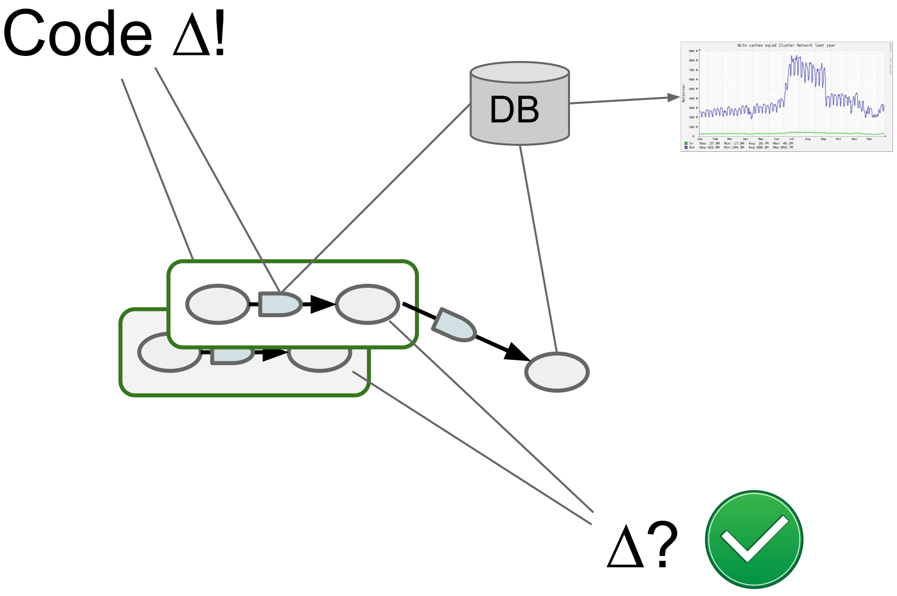
Obtaining quality metrics

- Processing tool (Spark/Hadoop) counters
  - Odd code path => bump counter
- Dedicated quality assessment pipelines
  - Reuse test oracle invariants in production

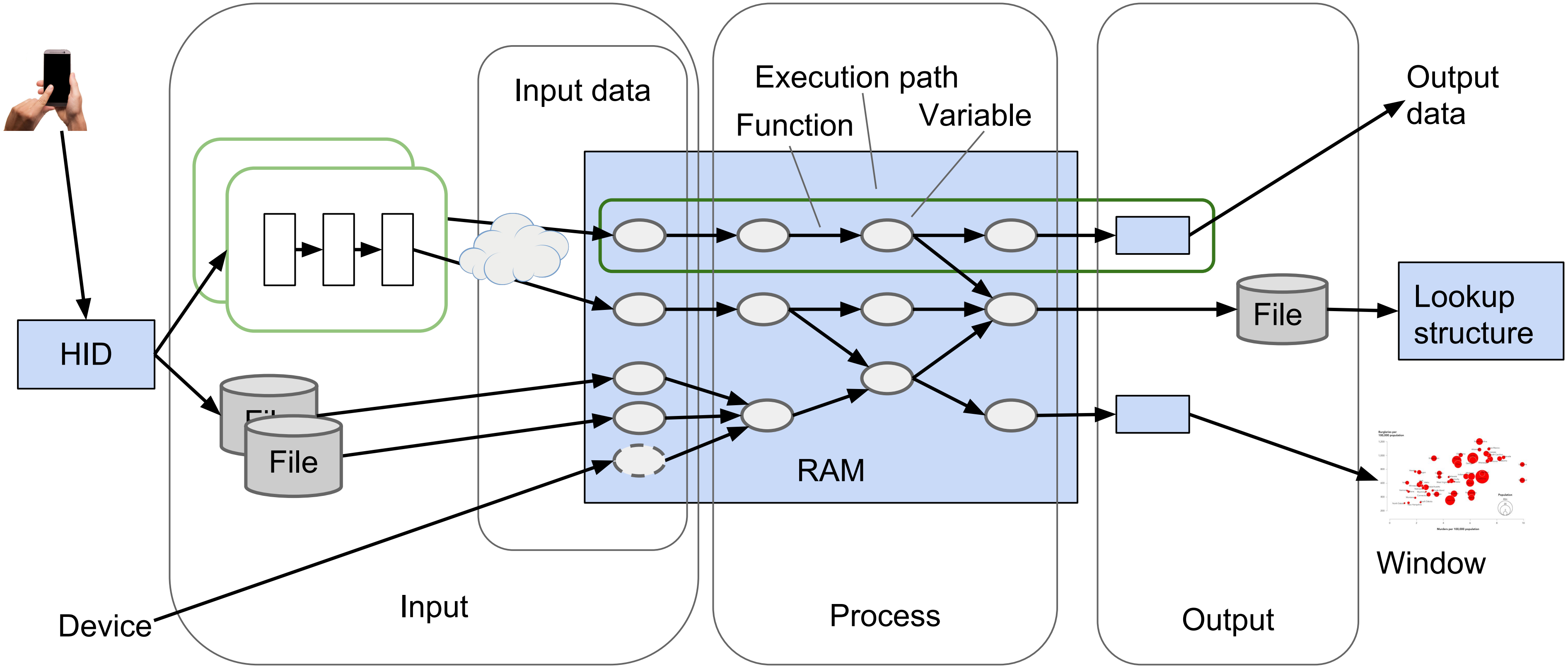


Quality testing in the process

- Binary self-contained
  - Validate in CI
- Relative vs history
  - E.g. large drops
  - Precondition for publishing dataset
- Push aggregates to DB
  - Standard ops: monitor, alert



Computer program anatomy



Data pipeline = yet another program

Don't veer from best practices

- Regression testing
- Design: Separation of concerns, modularity, etc
- Process: CI/CD, code review, static analysis tools
- Avoid anti-patterns: Global state, hard-coding location, duplication, ...

In data engineering, slipping is in the culture... :-(

- Mix in solid backend engineers, document "golden path"

Top anti-patterns

1. Test as afterthought or in production
   Data processing applications are suited for test!
2. Static test input in version control
3. Exact expected output test oracle
4. Unit testing volatile interfaces
5. Using mocks & dependency injection
6. Tool-specific test framework
7. Calling services / databases from (batch) jobs
8. Using wall clock time
9. Embedded fixture components

Further resources. Questions?

http://www.slideshare.net/lallea/data-pipelines-from-zero-to-solid

http://www.mapflat.com/lands/resources/reading-list

http://www.slideshare.net/mathieu-bastian/the-mechanics-of-testing-large-data-pipelines-qcon-london-2016

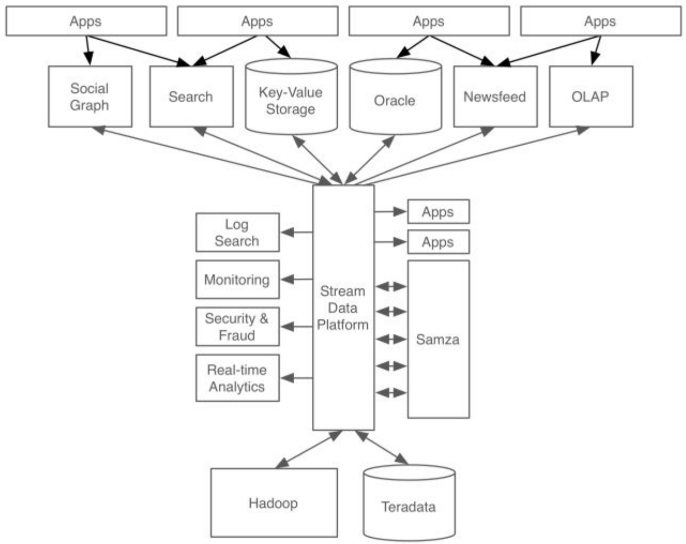
http://www.slideshare.net/hkarau/effective-testing-for-spark-programs-strata-ny-2015

https://spark-summit.org/2014/wp-content/uploads/2014/06/Testing-Spark-Best-Practices-Anupama-Shetty-Neil-Marshall.pdf

Bonus slides

Unified log

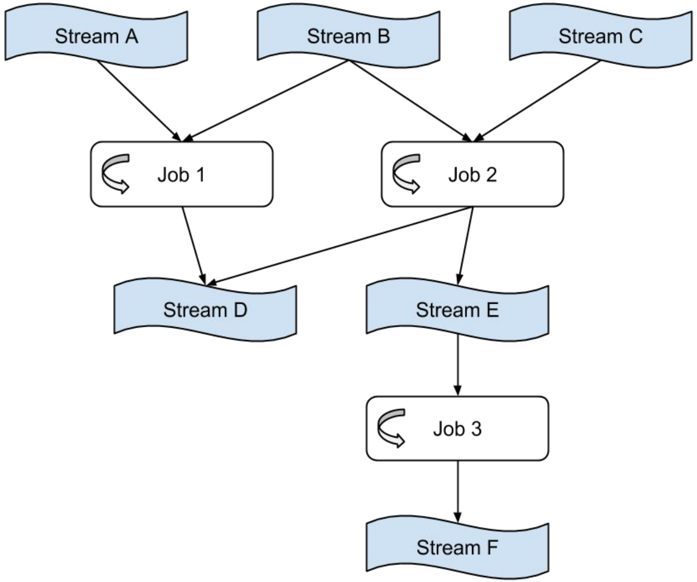
- Replicated append-only log
- Pub / sub *with history*
- Decoupled producers and consum
  - In source/deployment
  - In space
  - In time
- Recovers from link failures
- Replay on transformation bug fix



Credits: Confluent

Stream pipelines

- Parallelised jobs
- Read / write to Kafka
- View egress
  - Serving index
  - SQL / cubes for Analytics
- Stream egress
  - Services subscribe to topic
  - E.g. REST post for export

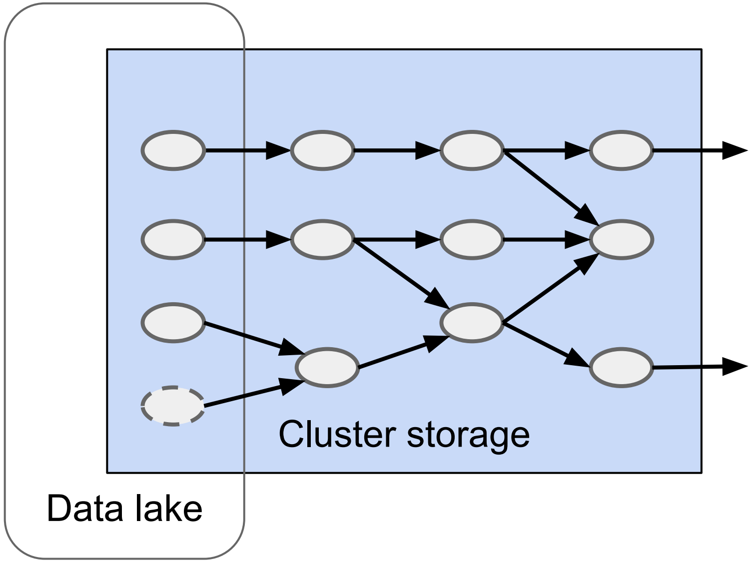


Credits: Apache Samza

The data lake

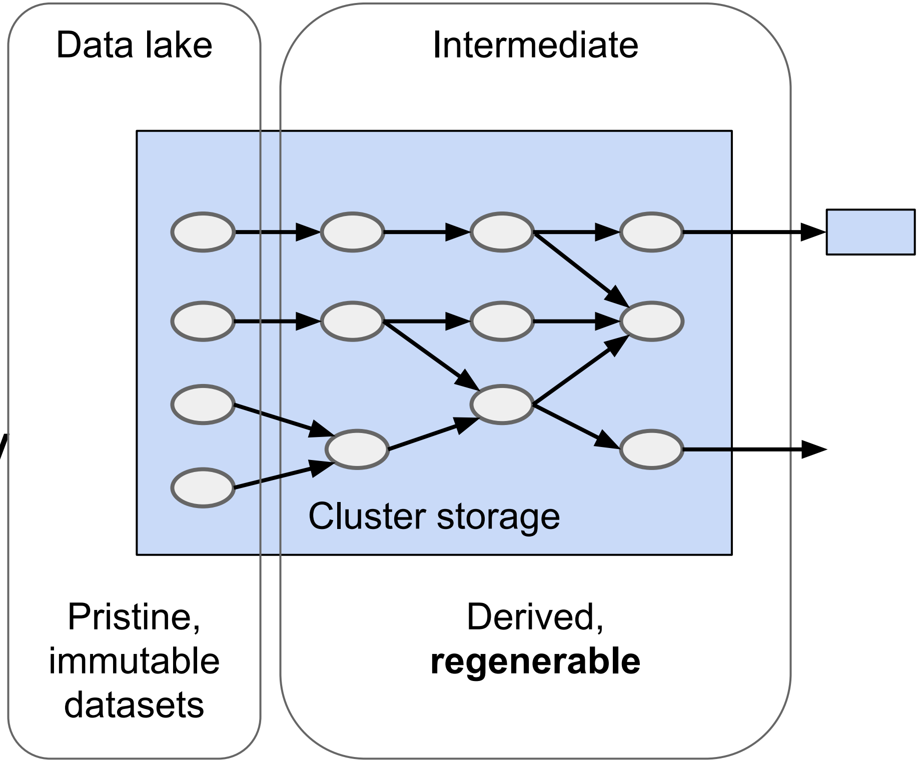
Unified log + snapshots

- Immutable datasets
- Raw, unprocessed
- Source of truth from batch processing perspective
- Kept as long as permitted
- Technically homogeneous
    - Except for raw imports



Batch pipelines

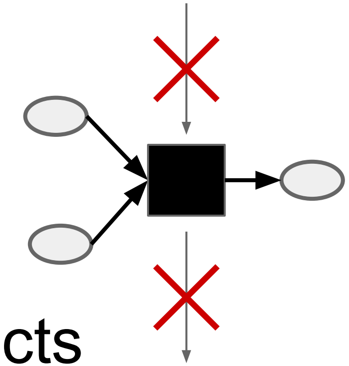
- Things will break
  - Input will be missing
  - Jobs will fail
  - Jobs will have bugs
- Datasets must be rebuilt
- Determinism, idempotency
- Backfill missing / failed
- Eventual correctness



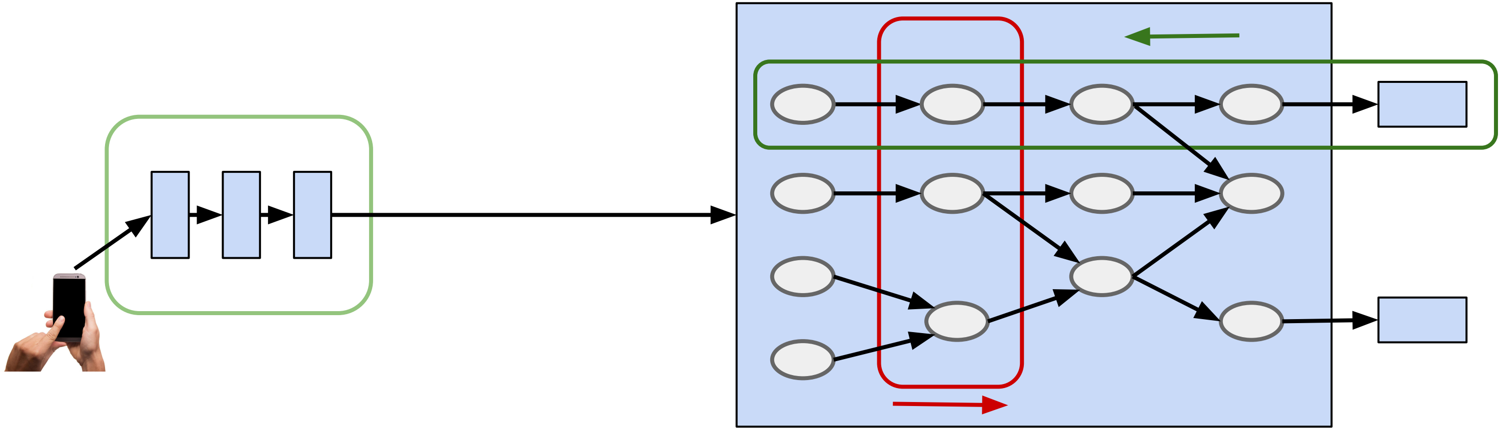
Batch job

Job == function([input datasets]): [output datasets]

- No orthogonal concerns
  - ~~Invocation~~
  - ~~Scheduling~~
  - ~~Input / output location~~
- Testable
- No other input factors, no side-effects
- Ideally: atomic, deterministic, idempotent
- **Necessary for audit**



Data pipelines

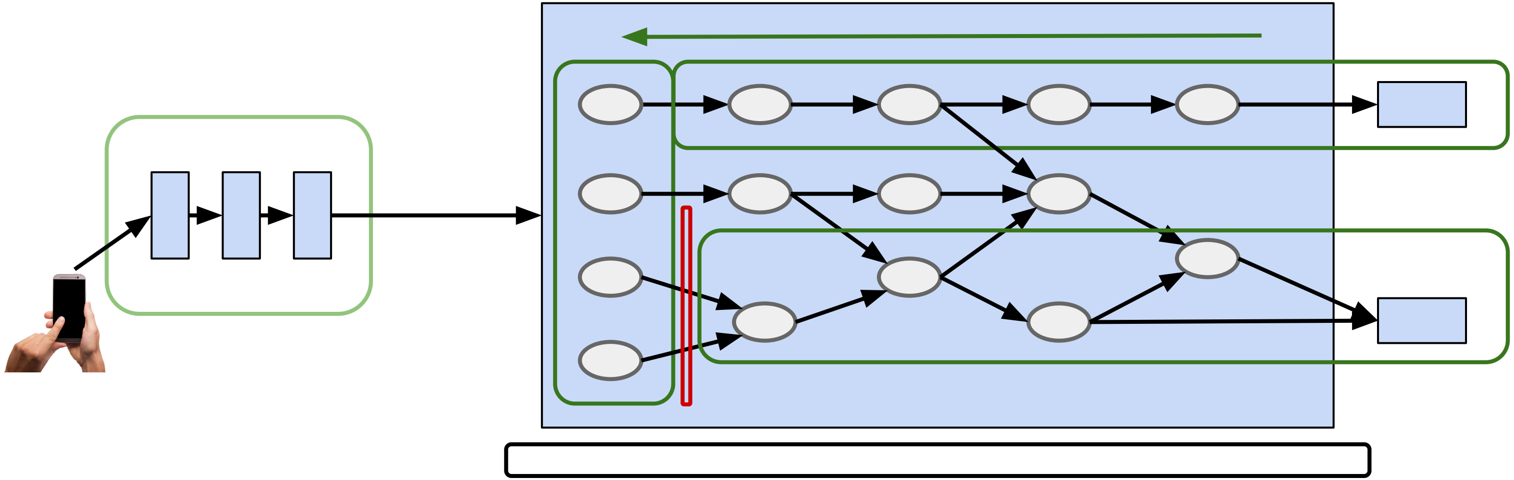


Form teams that are driven by business cases & need

Forward-oriented -> filters implicitly applied

Beware of: duplication, tech chaos/autonomy, privacy loss

Data platform, pipeline chains



Common data infrastructure

Productivity, privacy, end-to-end agility, complexity

Beware: producer-consumer disconnect

Ingress / egress representation

Larger variation:

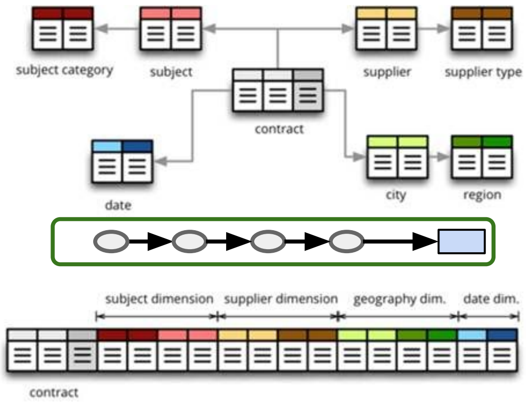
- Single file
- Relational database table
- Cassandra column family, other NoSQL
- BI tool storage
- BigQuery, Redshift, ...

Egress datasets are also atomic and immutable.

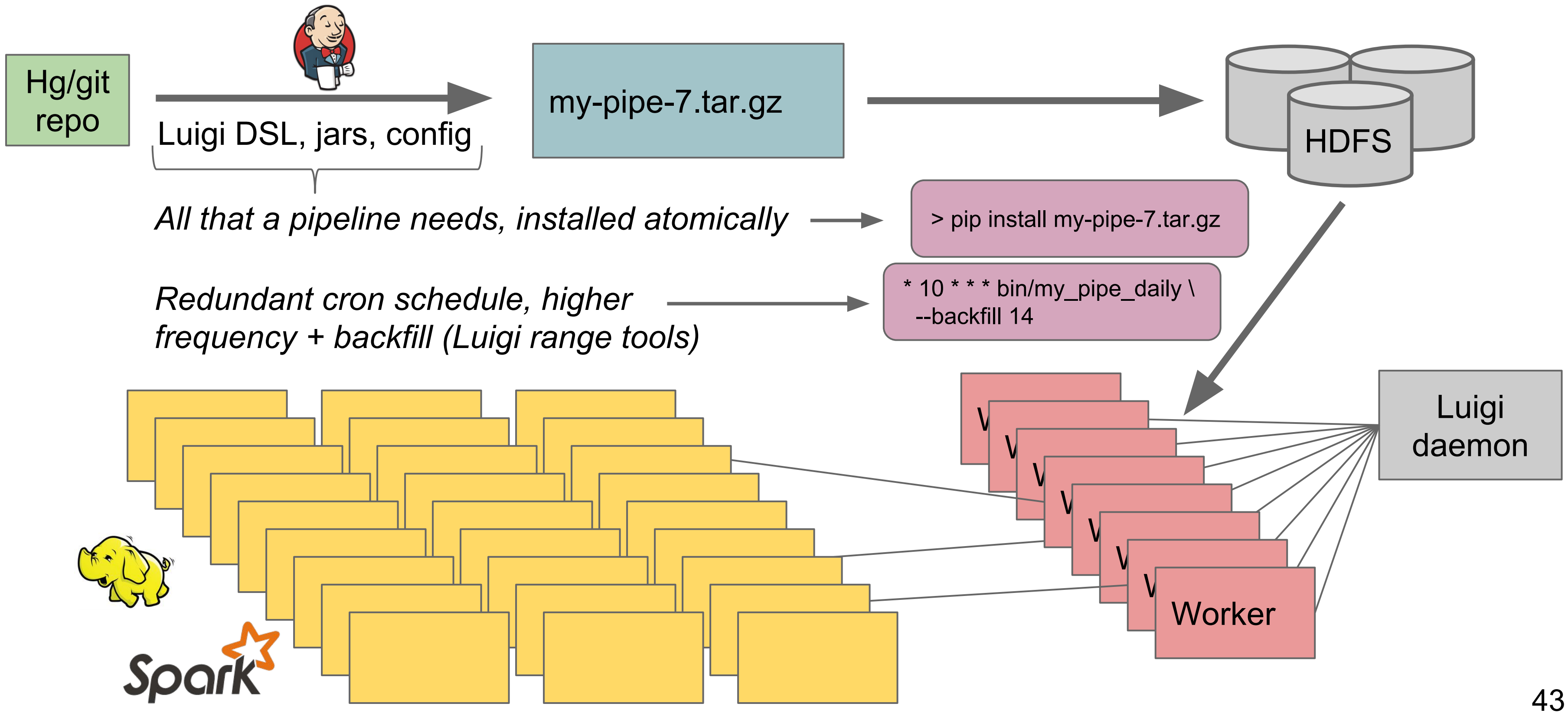
E.g. write full DB table / CF, switch service to use it, never change it.

Egress datasets

- Serving
  - Precomputed user query answers
  - Denormalised
  - **Cassandra**, (many)
- Export & Analytics
  - **SQL** (single node / Hive, Presto, ..)
  - Workbenches (Zeppelin)
  - (Elasticsearch, proprietary OLAP)
- BI / analytics tool needs change frequently
  - Prepare to redirect pipelines



Deployment


Hg/git repo
Luigi DSL, jars, config
my-pipe-7.tar.gz
HDFS
All that a pipeline needs, installed atomically
> pip install my-pipe-7.tar.gz
Redundant cron schedule, higher frequency + backfill (Luigi range tools)
* 10 * * * bin/my_pipe_daily \
--backfill 14
Luigi daemon
Worker
Spark