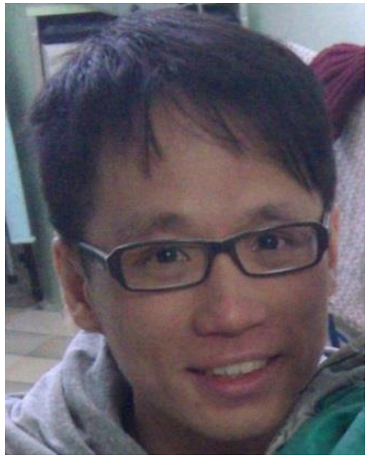
Leveraging Parallel Data Processing Frameworks with Verified Lifting



Maaz Ahmad

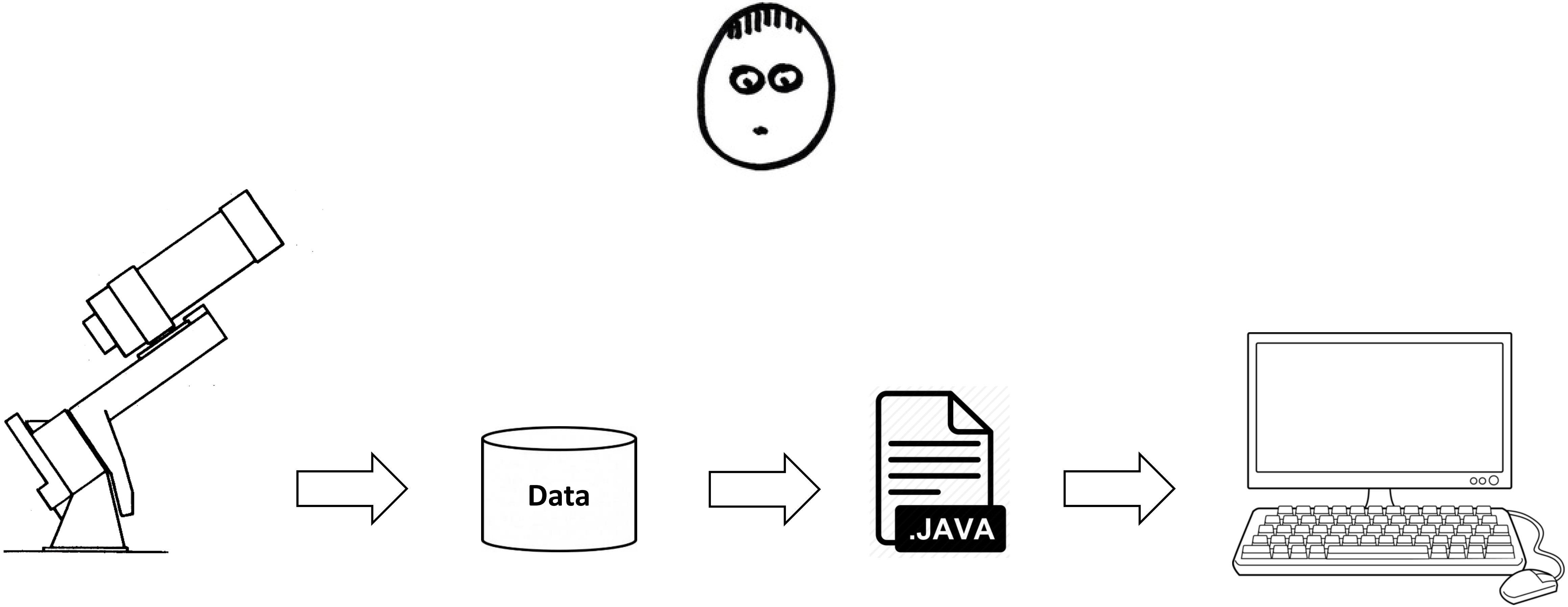
Alvin Cheung

PLSE

UW DB

UNIVERSITY of WASHINGTON

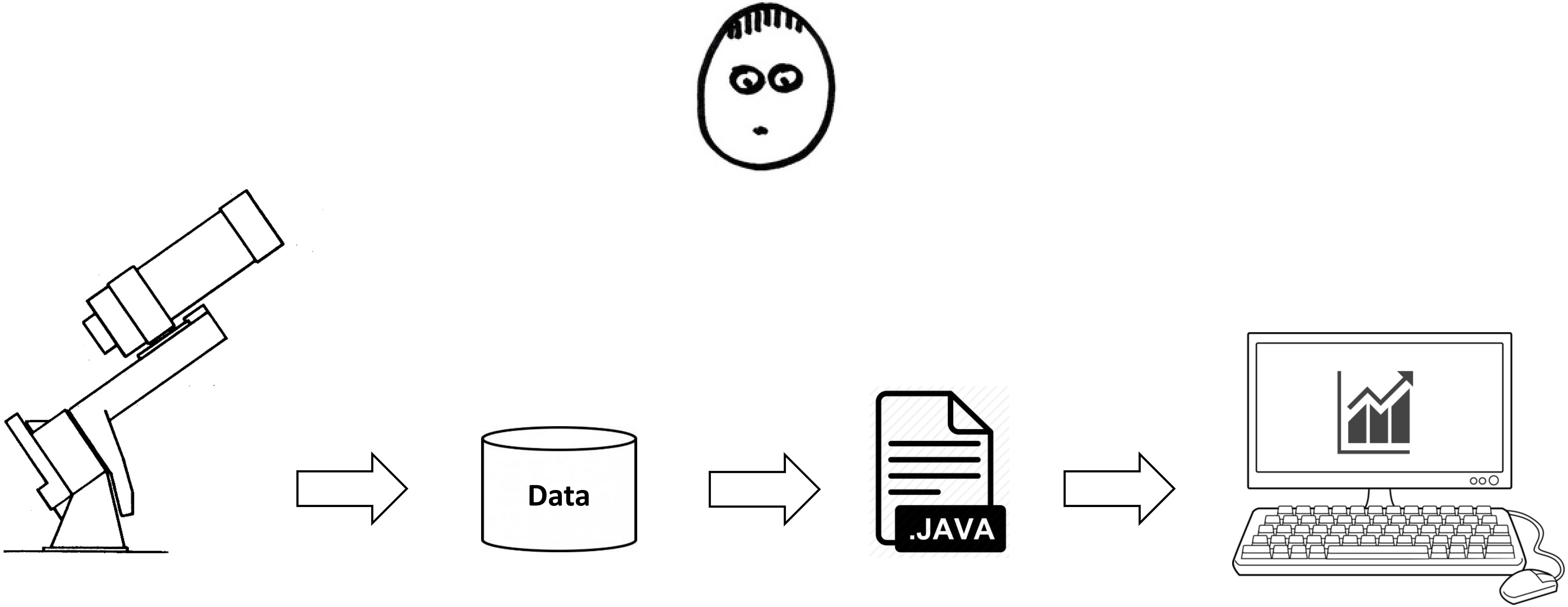
Motivation



Data Collection Tool

Data Analytics Application
(Sequential Java)

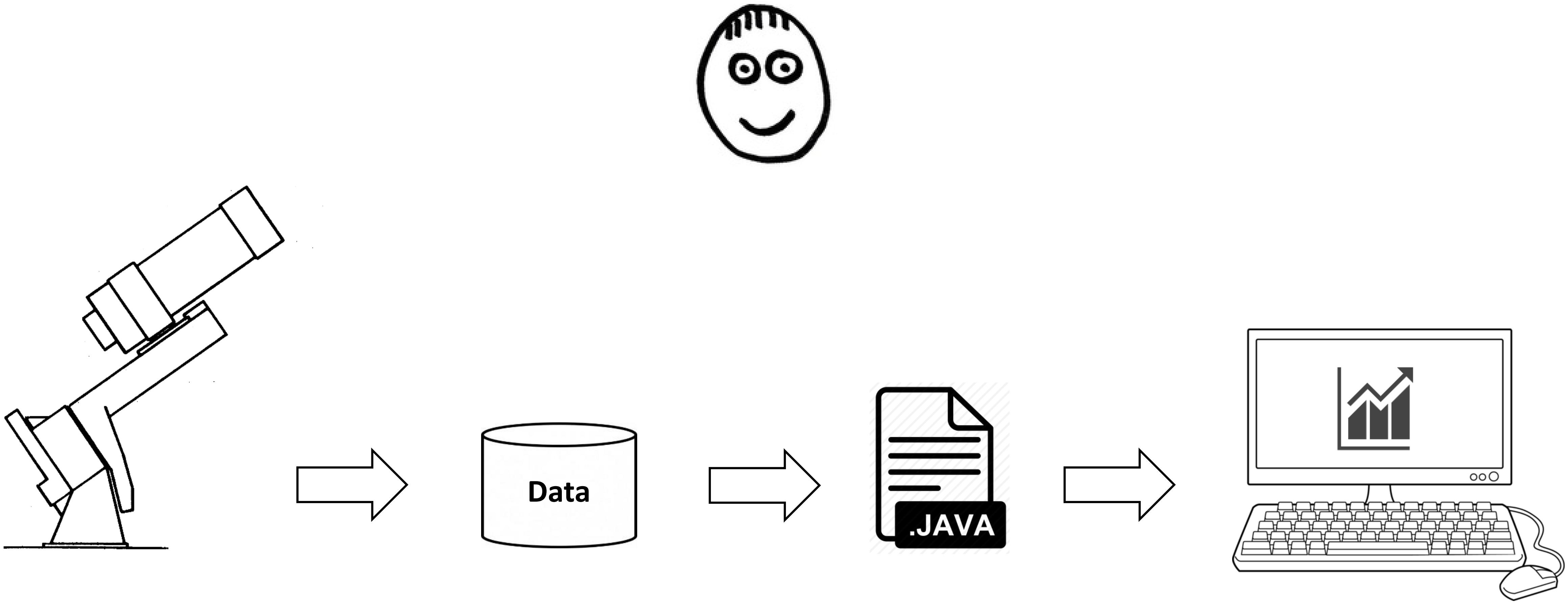
Motivation



Data Collection Tool

Data Analytics Application
(Sequential Java)

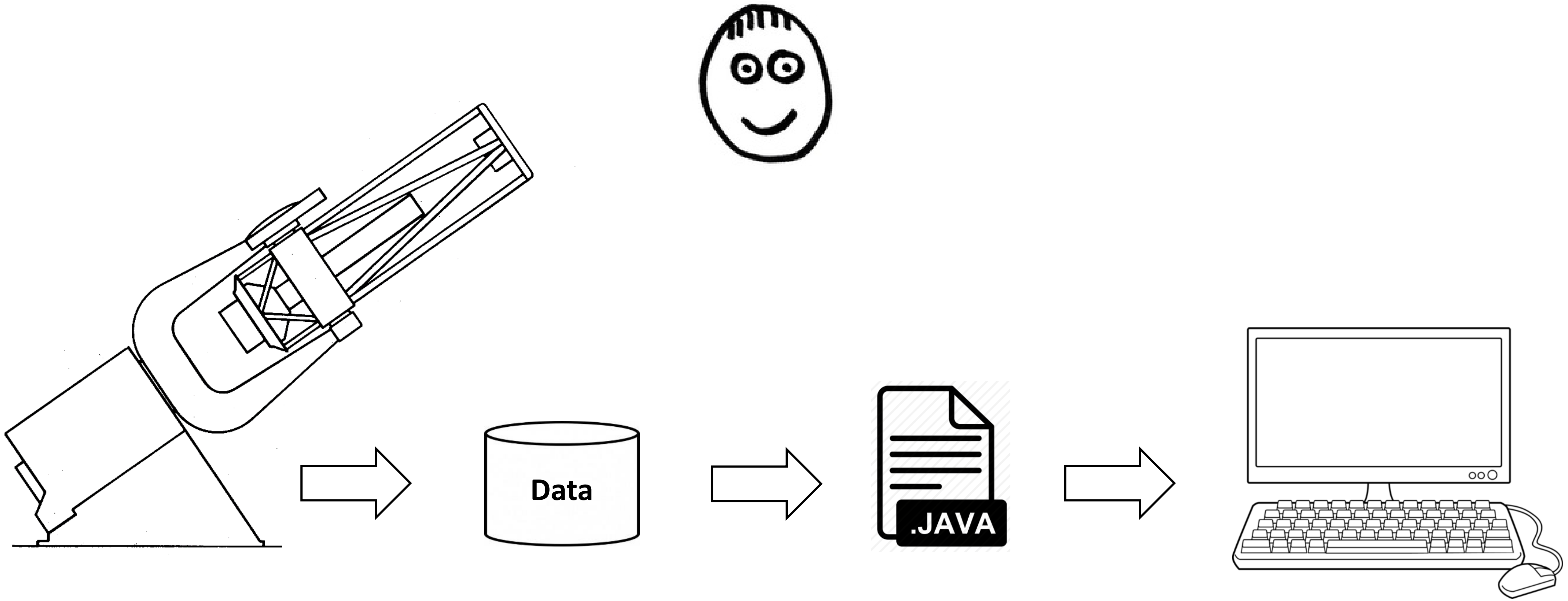
Motivation



Data Collection Tool

Data Analytics Application
(Sequential Java)

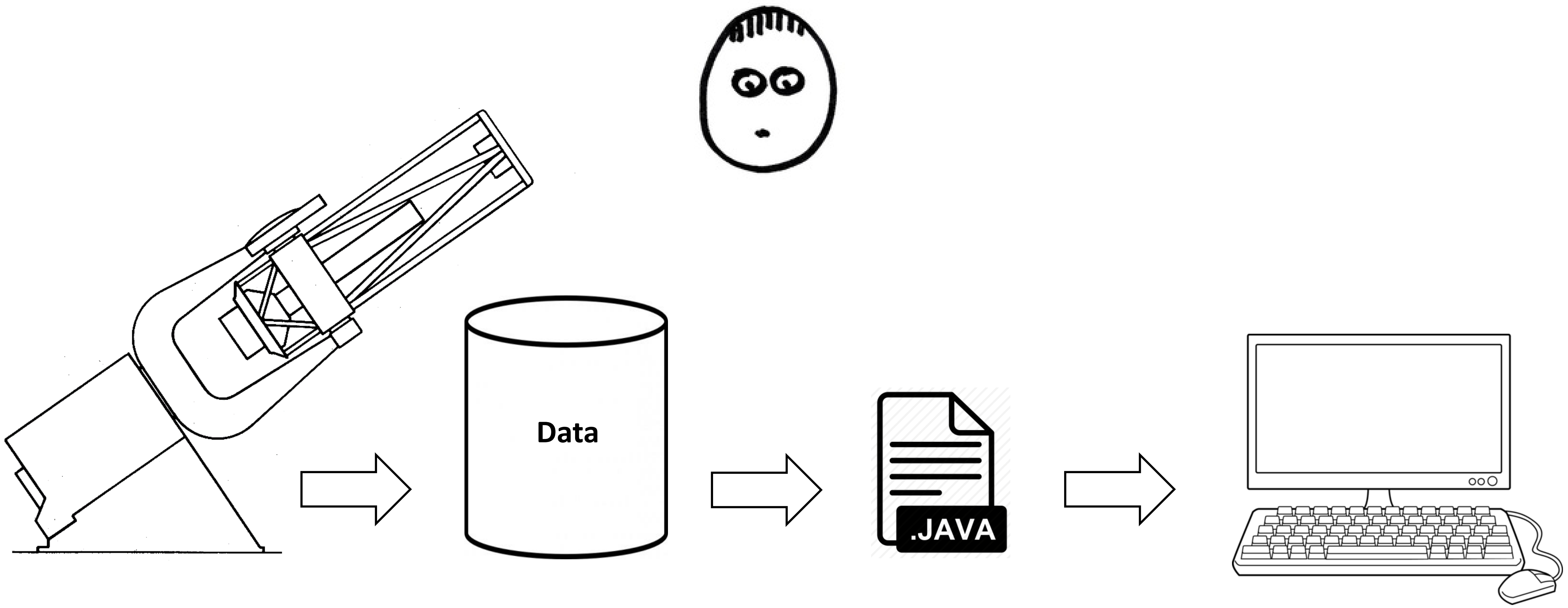
Motivation



Data Collection Tool

Data Analytics Application
(Sequential Java)

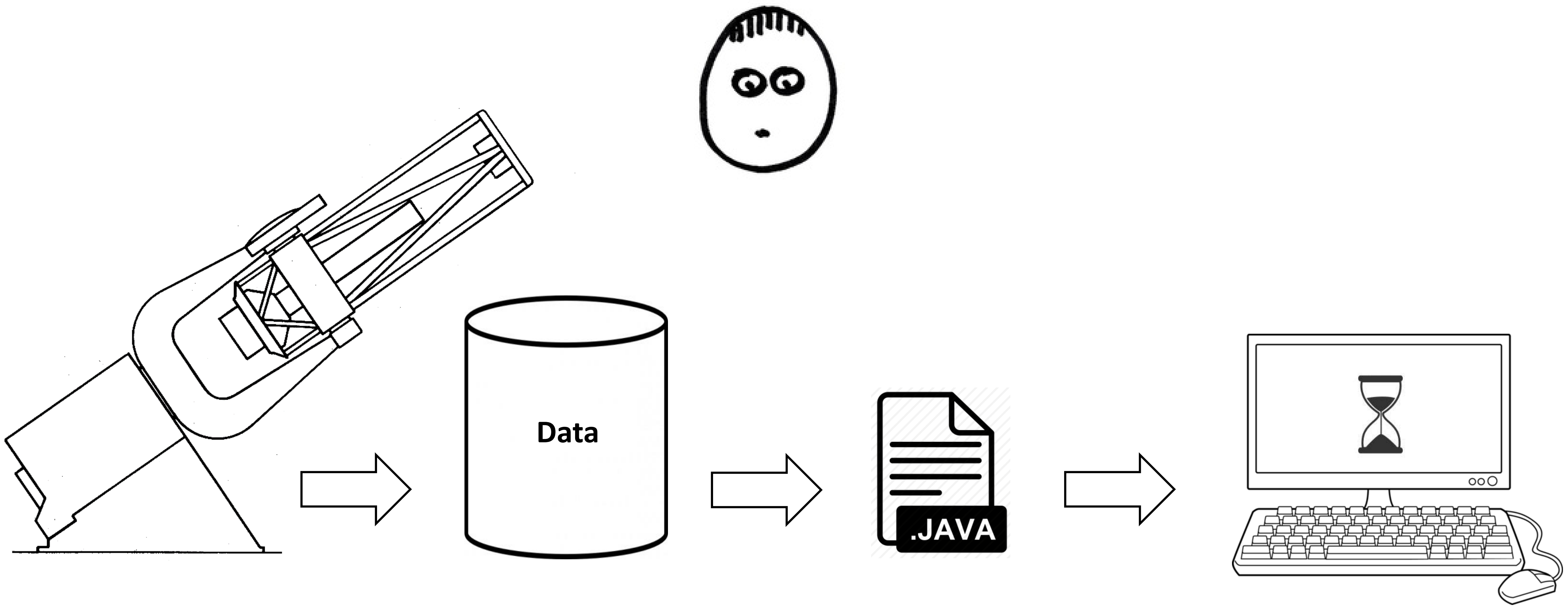
Motivation



Data Collection Tool

Data Analytics Application
(Sequential Java)

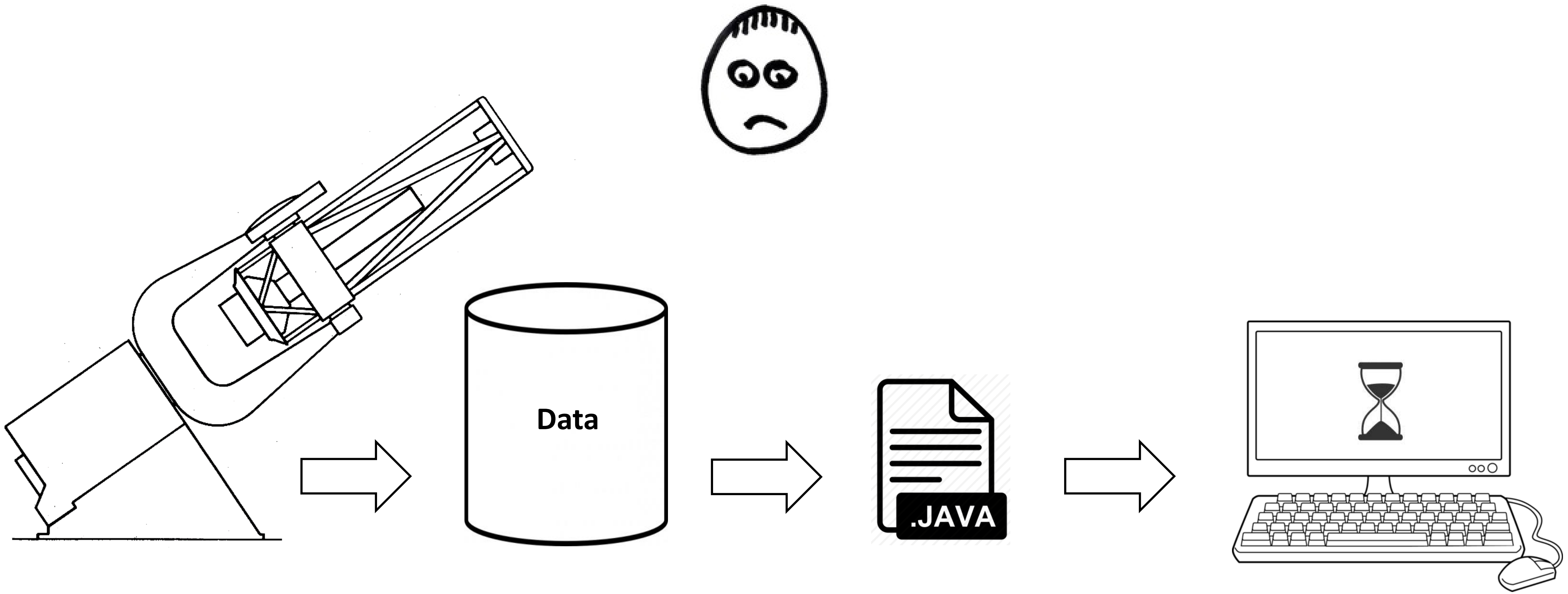
Motivation



Data Collection Tool

Data Analytics Application
(Sequential Java)

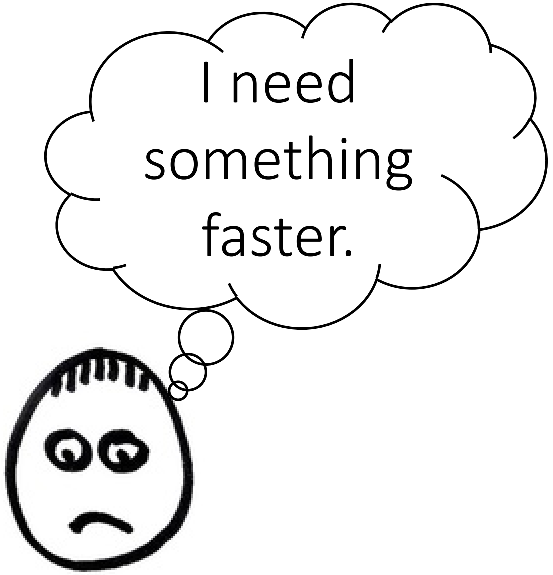
Motivation

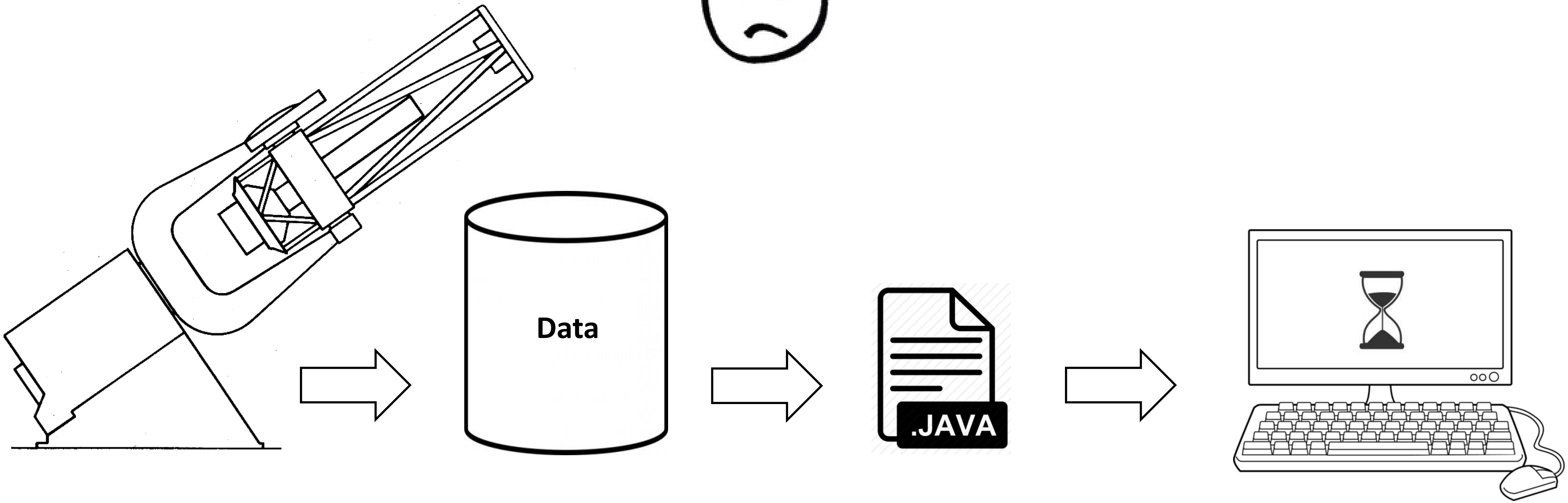


Data Collection Tool

Data Analytics Application
(Sequential Java)

Motivation





Data Collection Tool

Data Analytics Application
(Sequential Java)

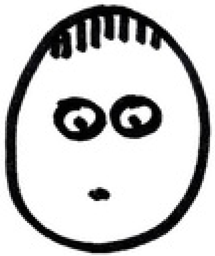
Parallel Processing Frameworks

Spark

GraphLab

hadoop

APACHE STORM™



OpenMP®

Flink

Parallel Processing Frameworks

Spark

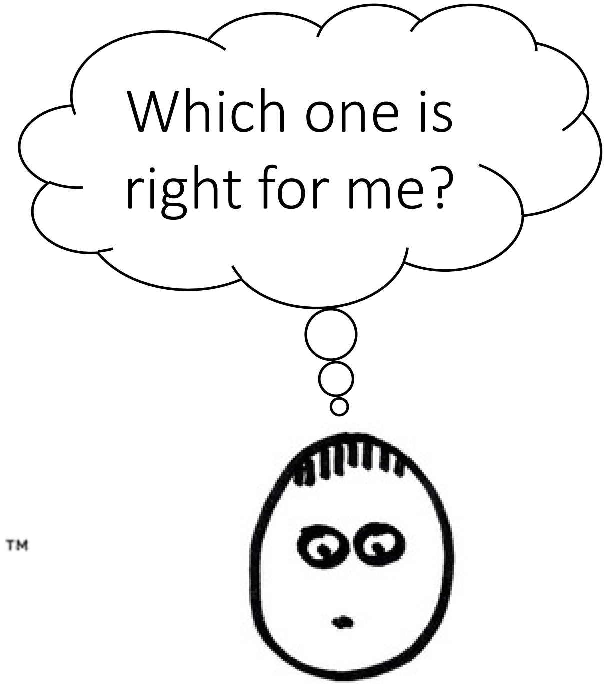
GraphLab

hadoop

APACHE STORM™

OpenMP®

Flink



Parallel Processing Frameworks

Spark

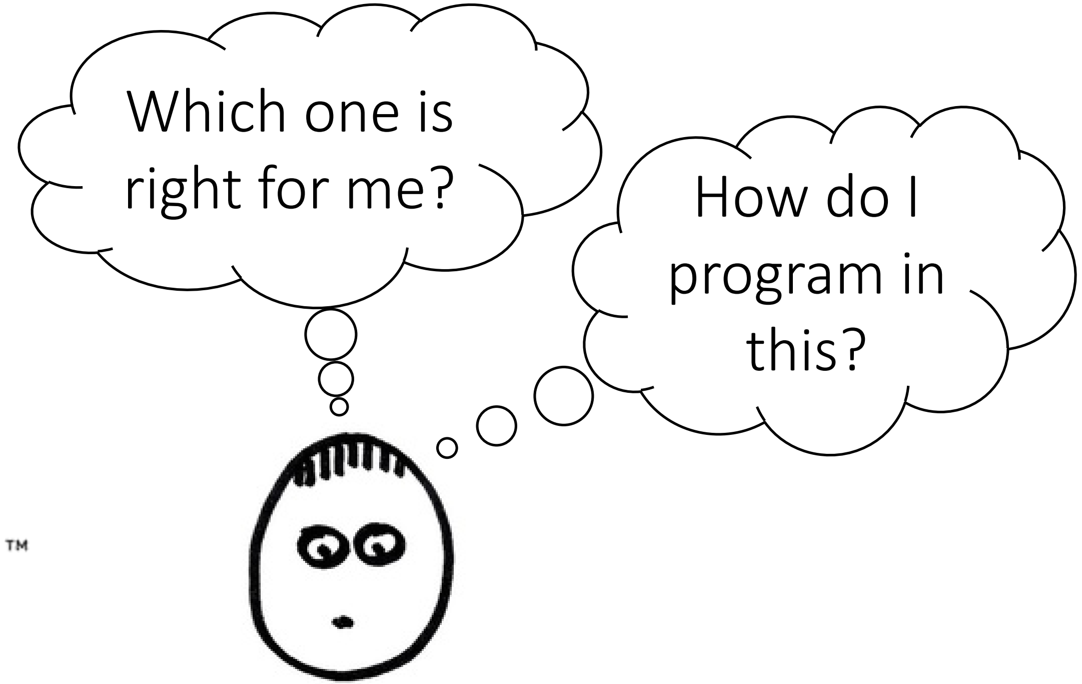
GraphLab

hadoop

APACHE STORM™

OpenMP®

Flink



Parallel Processing Frameworks

Spark

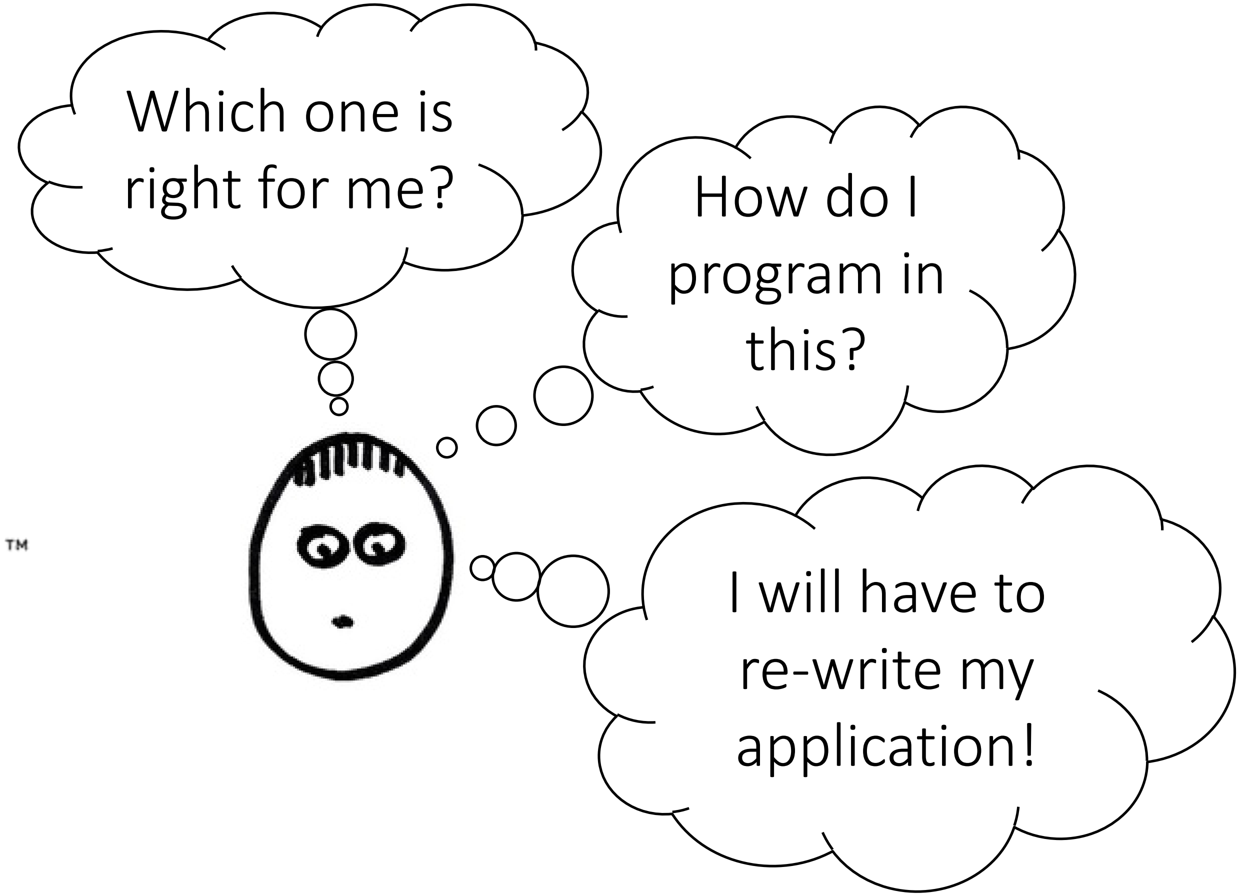
GraphLab

hadoop

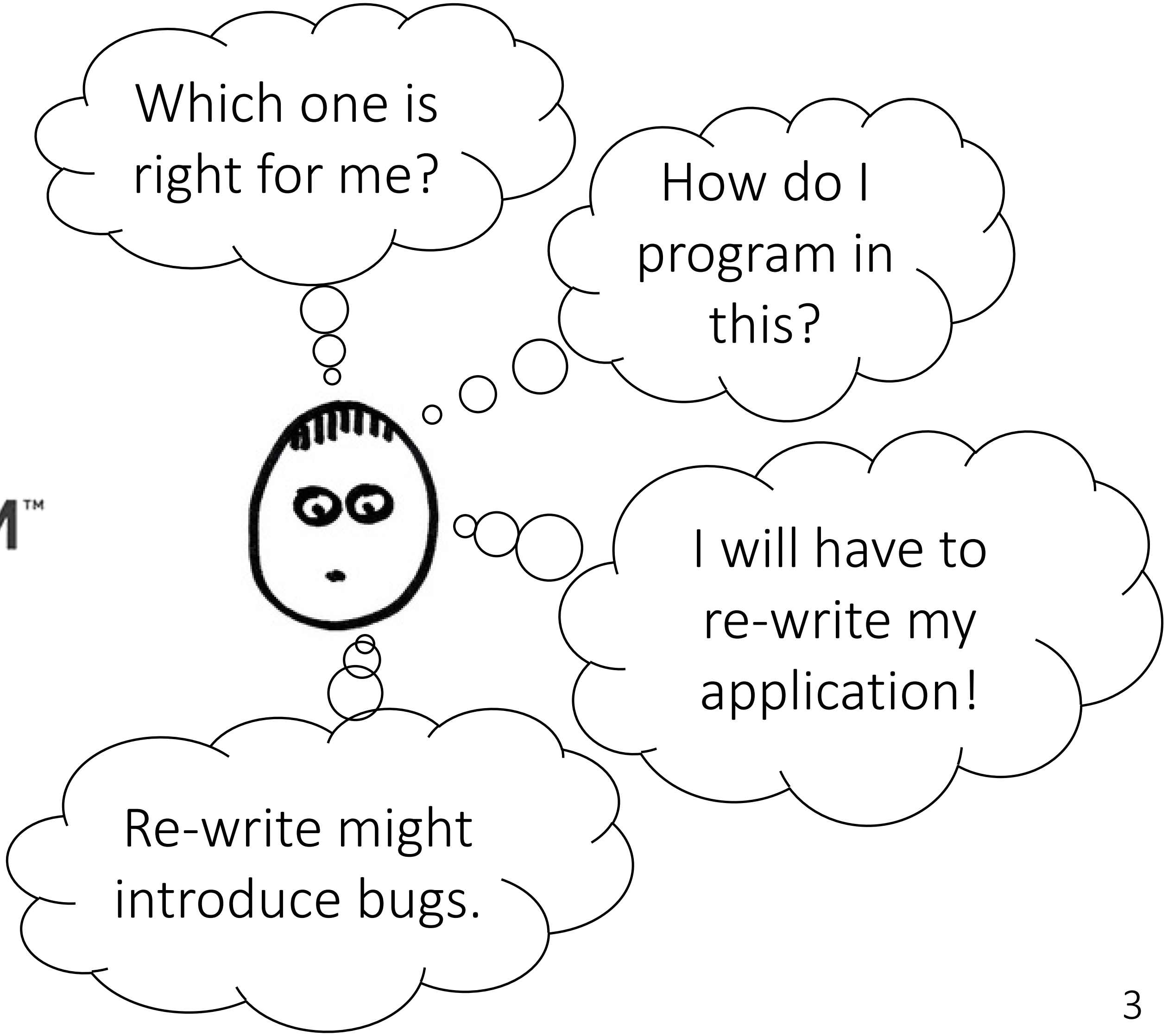
APACHE STORM™

OpenMP®

Flink

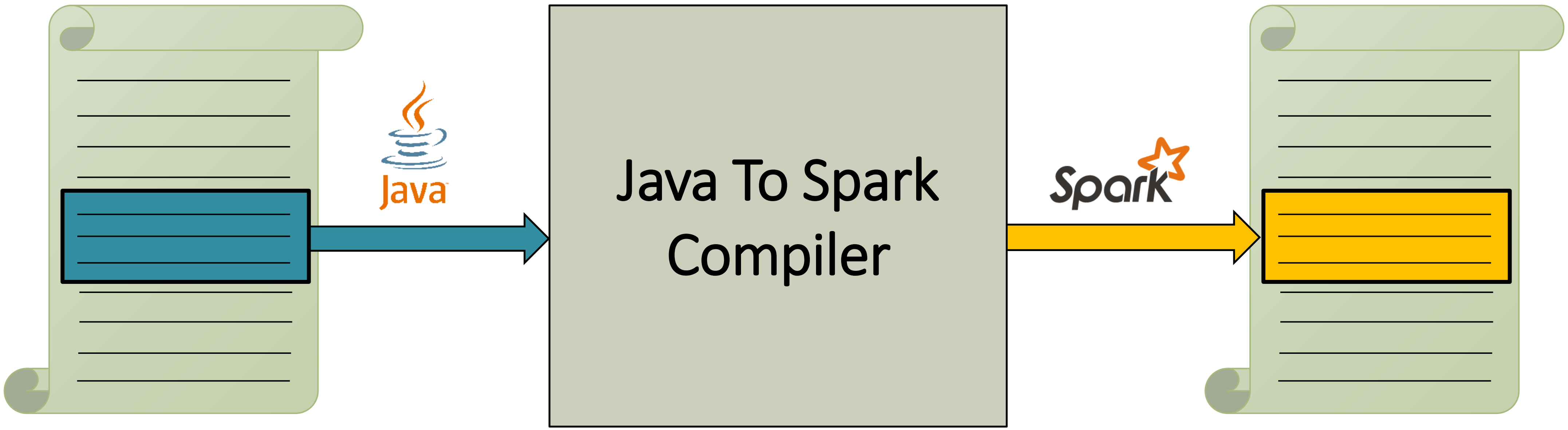


Parallel Processing Frameworks



How can we make life easier?

How can we make life easier?



```
for(int i = 0; i < data; i++){
    ...
}
```

Syntax Directed Rules

```
mapper(key, data){
    ...
}
reducer(key, values){
    ...
}
```

Hard to come up with rules

Brittle to code pattern changes

$f_m(val) \rightarrow \boxed{\dots}$

$f_r(val1, val2) \rightarrow \boxed{\dots}$

output = reduce(map(data, f_m), f_r);

Syntax Directed Rules

Java

```
for(int i = 0; i < data; i++){
    ...
}
```

Syntax Directed Rules

hadoop

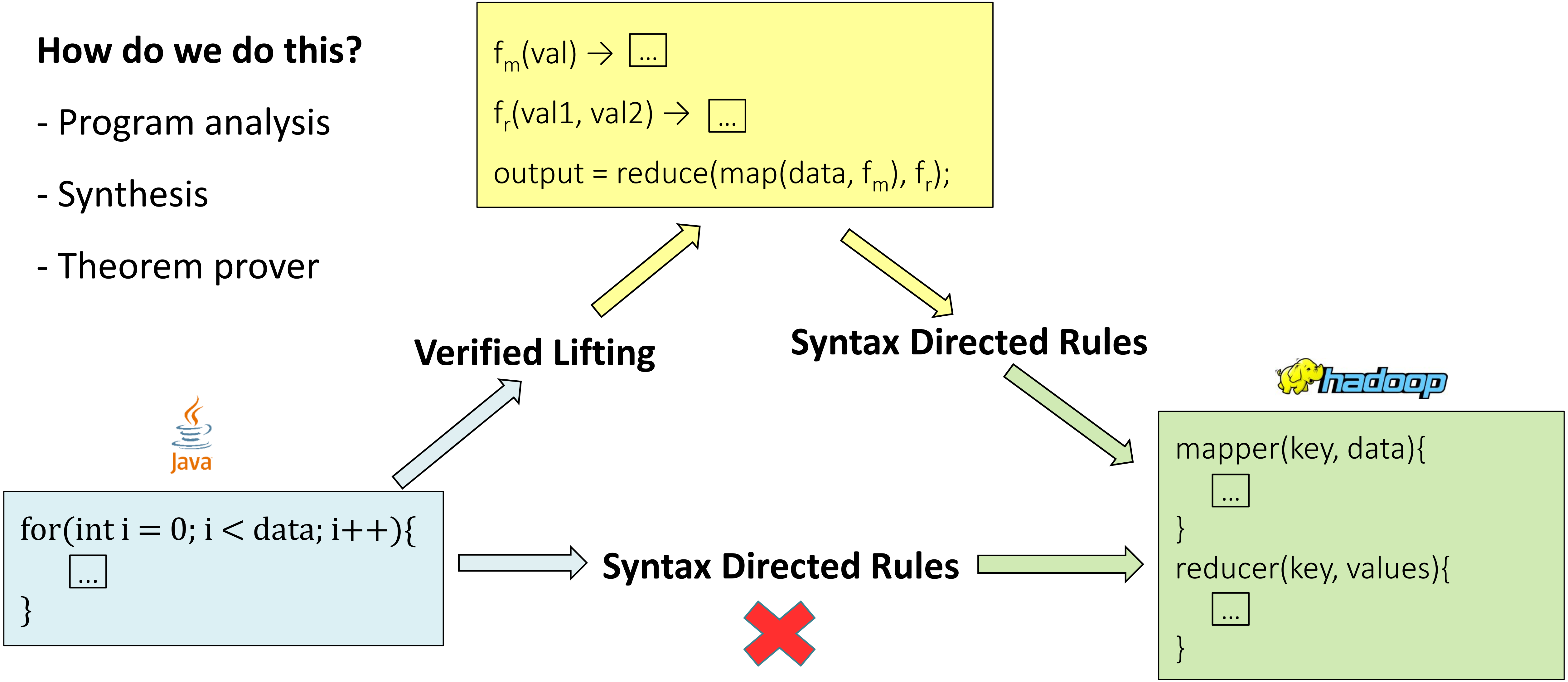
```
mapper(key, data){
    ...
}
reducer(key, values){
    ...
}
```

Hard to come up with rules

Brittle to code pattern changes

How do we do this?

- Program analysis
- Synthesis
- Theorem prover



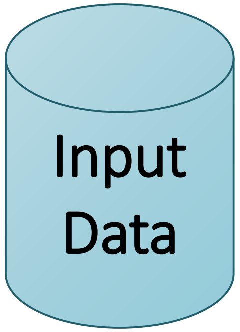
Hard to come up with rules

Brittle to code pattern changes

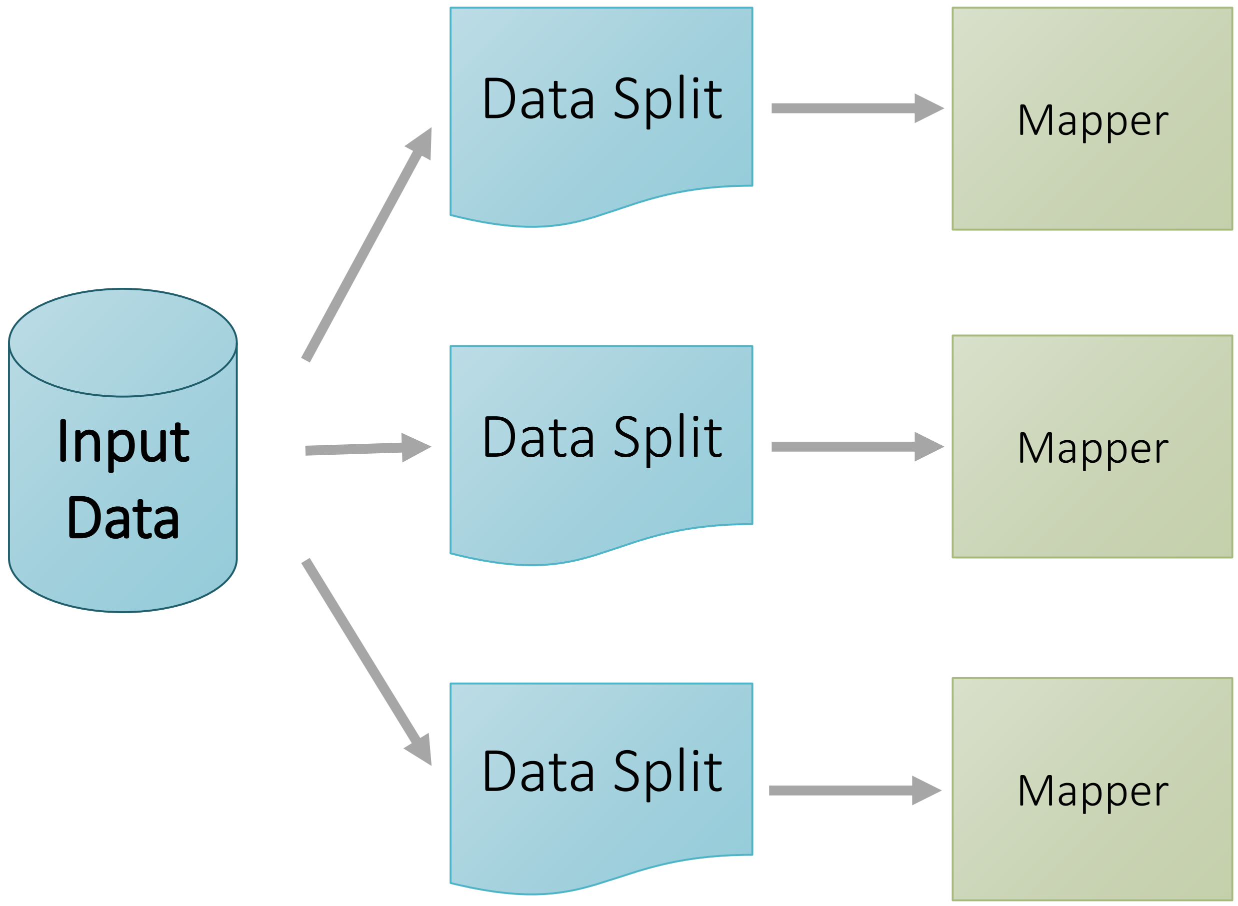
Introducing CASPER

- Re-targets sequential Java code fragments to Hadoop/Spark frameworks.
- **Input:** Unannotated sequential Java application source code.
- **Output:** Translated application source code that runs on top of Hadoop/Spark to leverage its parallel execution.

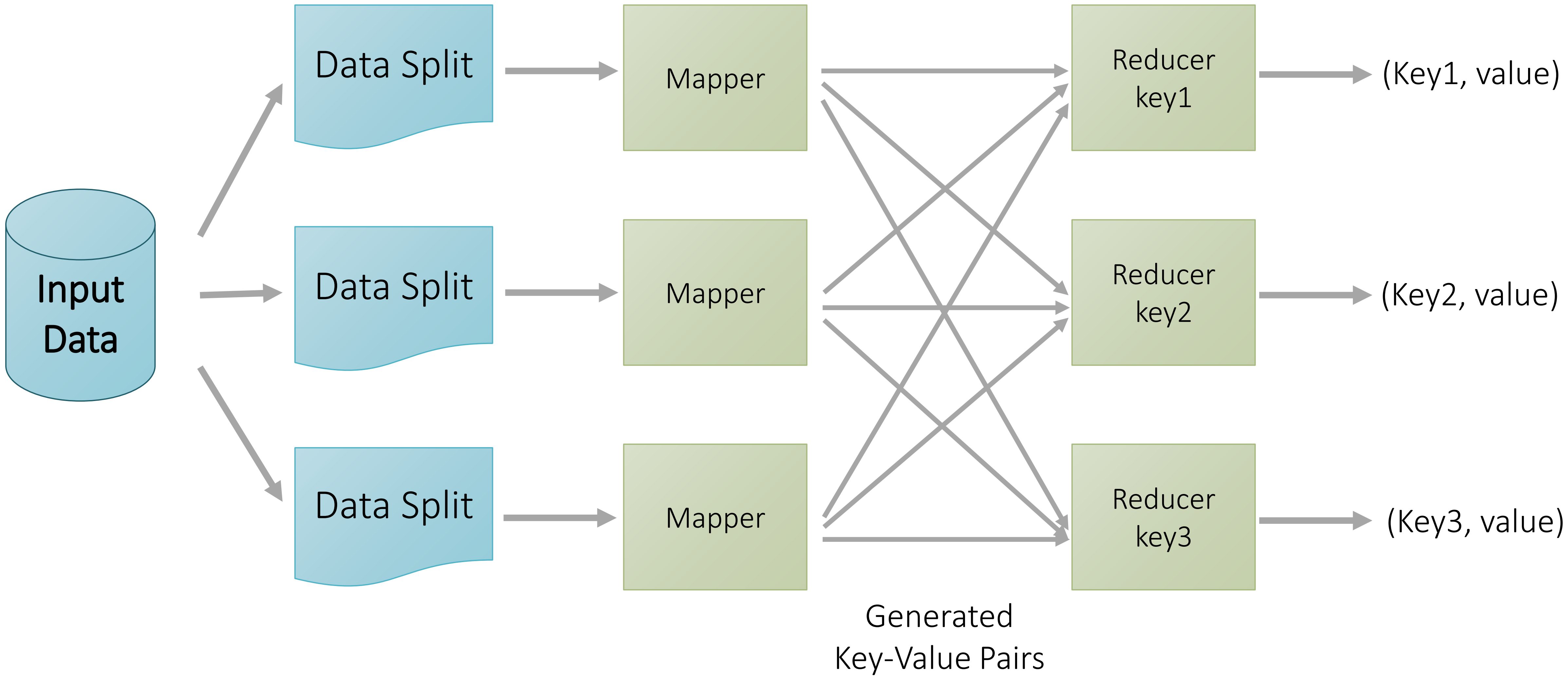
MapReduce Overview



MapReduce Overview



MapReduce Overview



Verified Lifting

- Infer code semantics (summary) in a high level specification
- A summary describes the effect of code on the output variables

Java Code Fragment

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

Summary

$$data\_sqr \equiv \sum_{i=0}^{i=data.size()-1} data[i]^2$$

Verified Lifting

- Infer code semantics (summary) in a high level specification
- A summary describes the effect of code on the output variables

Java Code Fragment

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

Post-condition

Summary

$$data\_sqr \equiv \sum_{i=0}^{i=data.size()-1} data[i]^2$$

Verified Lifting

- Infer code semantics (summary) in a high level specification
- A summary describes the effect of code on the output variables

Java Code Fragment

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

Post-condition

Summary

$$\boxed{data\_sqr} \equiv \sum_{i=0}^{i=data.size()-1} data[i]^2$$

Verified Lifting

- Infer code semantics (summary) in a high level specification
- A summary describes the effect of code on the output variables

Java Code Fragment

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

Post-condition

Summary

$$\boxed{data\_sqr} \equiv \sum_{i=0}^{i=data.size()-1} data[i]^2$$

- Specifications must be trivial to translate.
- Program specification exhibits good parallelism.

Code Summaries in Casper

Code Summaries in Casper

$\forall v \in outputVariables.$

Code Summaries in Casper

$$\forall v \in outputVariables. \quad v \equiv f_{reduce}(v_0, reduce(map(data, f_{map}), f_{reduce}))$$

Code Summaries in Casper

$$\forall v \in outputVariables. \quad v \equiv f_{reduce}(v_0, reduce(map(data, f_{map}), f_{reduce}))$$

Where,

$map$and $f\,reduce$ are synthesized for each code fragment.

ons.

$f\ map$ $map\,p\,map$ and f_{reduce} are synthesized for each code fragment.

Restricting Search Space

- Use Syntax-Guided Synthesis (SyGuS) to generate f_{map} and f_{reduce}.
- Use a grammar to specify a set of candidate summaries.
- Grammar is dynamically generated for each code fragment.

Grammar Generation: f_{map}

- The body of f_{map} is just a sequence of emits.
  - Begin with number of emits equal to number of output variables.
  - Incrementally add emits statements up to a user-defined bound.

$$Map \;\rightarrow Map\ Map \mid Emit$$

$$Emit \;\rightarrow\; emit(Key, Value);\ \mid\ if(Condition)\ emit(Key, Value);$$

$$Key \rightarrow IntExp \mid StringExp \mid BoolExp \mid\ \ldots$$

$$Value \rightarrow IntExp \mid StringExp \mid BoolExp \mid\ \ldots$$

Grammar Generation: f_{map}

- The key and value for each emit are generated using expression grammars.

Java Code Fragment

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

Integer Expression Grammar

$$IntExp \rightarrow IntExp + IntExp \mid IntExp * IntExp \mid data[IntExp] \mid IntVal$$

$$IntVal \rightarrow data\_sqr \mid i \mid literal$$

Grammar Generation: f_{reduce}

- The body of f_{reduce} implements a fold operation.

Java Code Fragment

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

Fold Expression Grammar

$$Reduce \rightarrow int\ res = literal;\ for(value : values)\{ res = FoldExp; \}\ emit(key, res);$$

$$FoldExp \rightarrow FoldExp + FoldExp \mid FoldExp * FoldExp \mid IntVal$$

$$IntVal \rightarrow res \mid val \mid key \mid literal$$

Verifying Equivalence

- CASPER uses Hoare-style verification conditions.
- Verification conditions are the *weakest pre-conditions* for the post-condition (code summary) to hold.
- Proving post-conditions for code fragments containing loops requires loop-invariants.

Verifying Equivalence Pt. 2

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

Verifying Equivalence Pt. 2

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

$$preCondition \equiv data\_sqr = 0$$

Verifying Equivalence Pt. 2

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

$$preCondition \equiv data\_sqr = 0$$

$$postCondition \equiv data\_sqr = reduce(map(data, f_{map}), f_{reduce})$$

Verifying Equivalence Pt. 2

```
data_sqr = 0;
for(int i = 0; i < data.size(); i++){
    data_sqr += data[i] * data[i];
}
```

$$preCondition \equiv data\_sqr = 0$$

$$postCondition \equiv data\_sqr = reduce(map(data, f_{map}), f_{reduce})$$

$$loopInvariant \equiv data\_sqr = reduce(map(data[0..i], f_{map}), f_{reduce}) \wedge 0 \leq i \leq data.length$$

Formal Verification

- We have modelled the MapReduce library in Dafny.
- The generated summary is compiled down to Dafny code.
- Code annotations are automatically generated. These include:
  - Verification conditions
  - Proof lemmas

Lemma Example

lemma InductiveStep (data: seq<int>, i: int, data_sqr: int)

requires invariant(data, i, data_sqr) && i < |data|

ensures invariant(data, i + 1, data_sqr + (data[i] * data[i]));

{

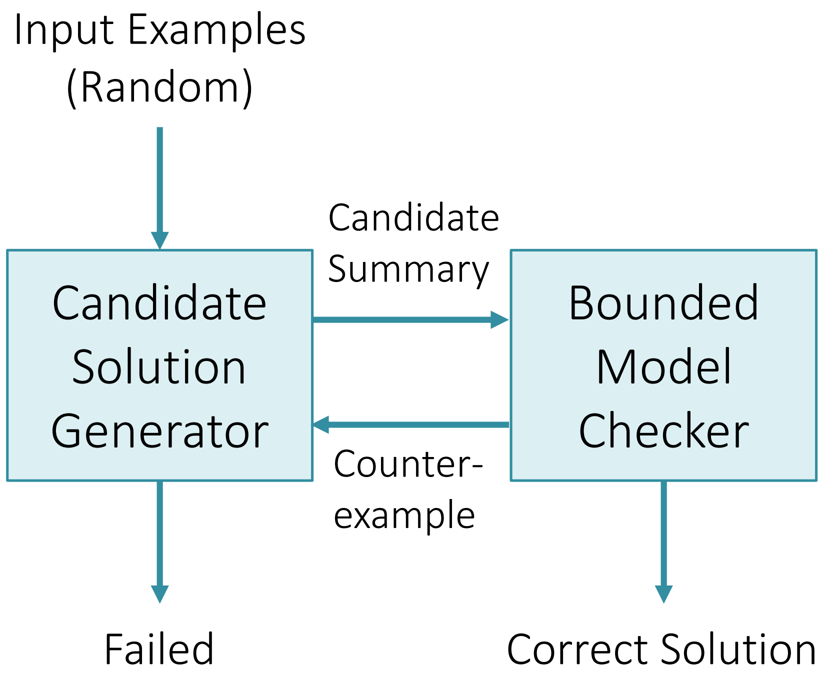
assert map (data, i+1) == f_{map}(data, i) + map(data, i);

assert f_{reduce}(f_{map}(data, i), 0) == data[i] * data[i];

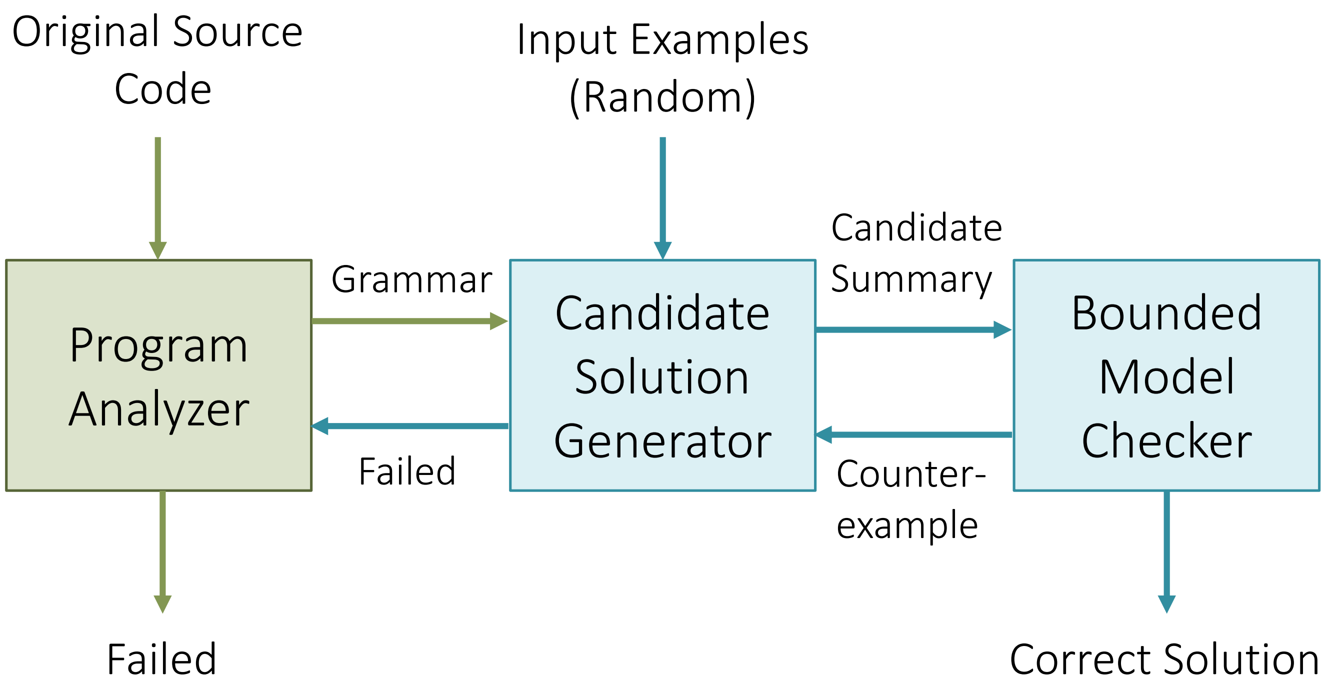
...

}

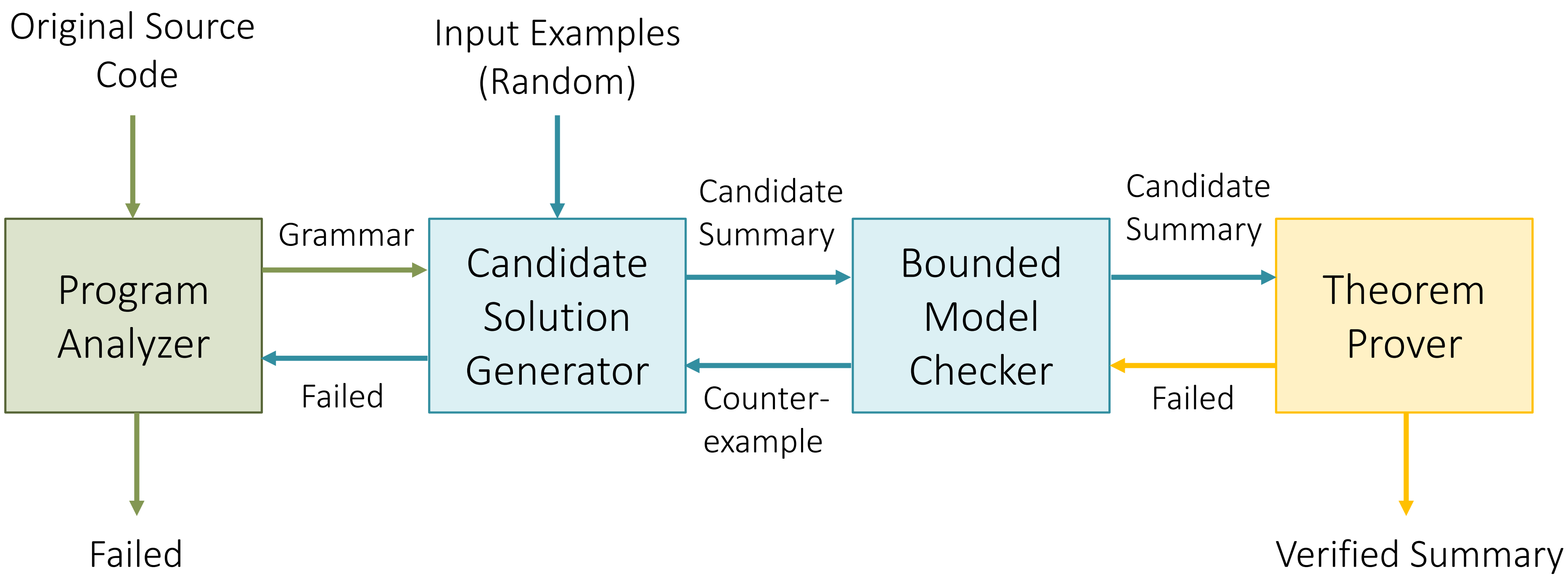
CASPER Architecture Diagram



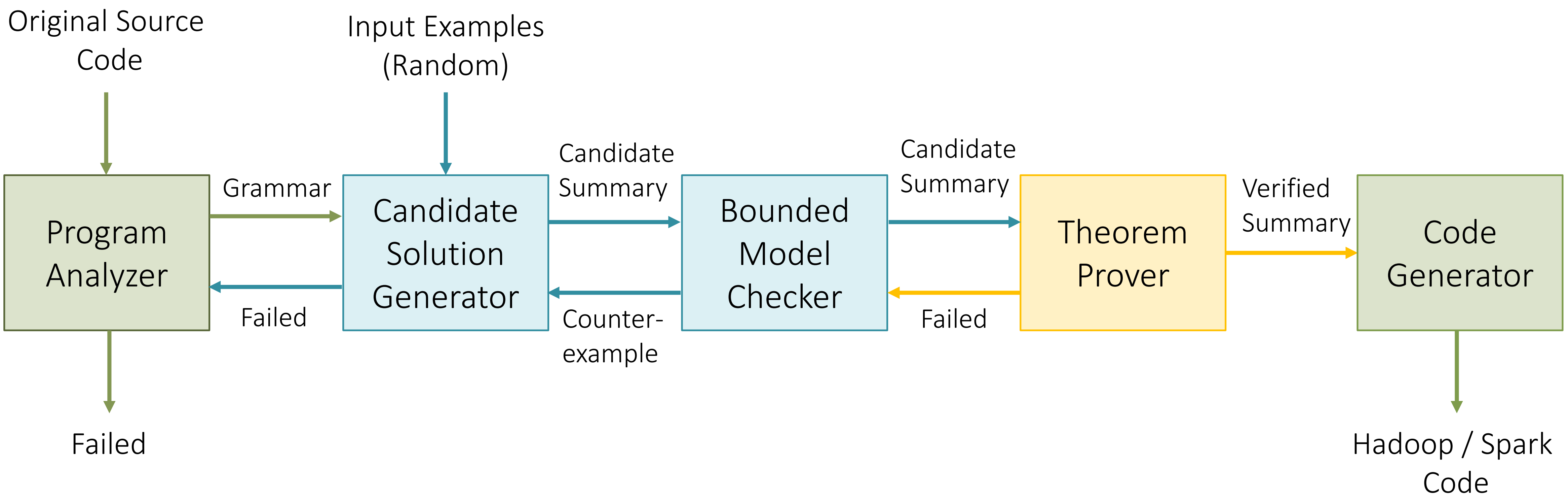
CASPER Architecture Diagram



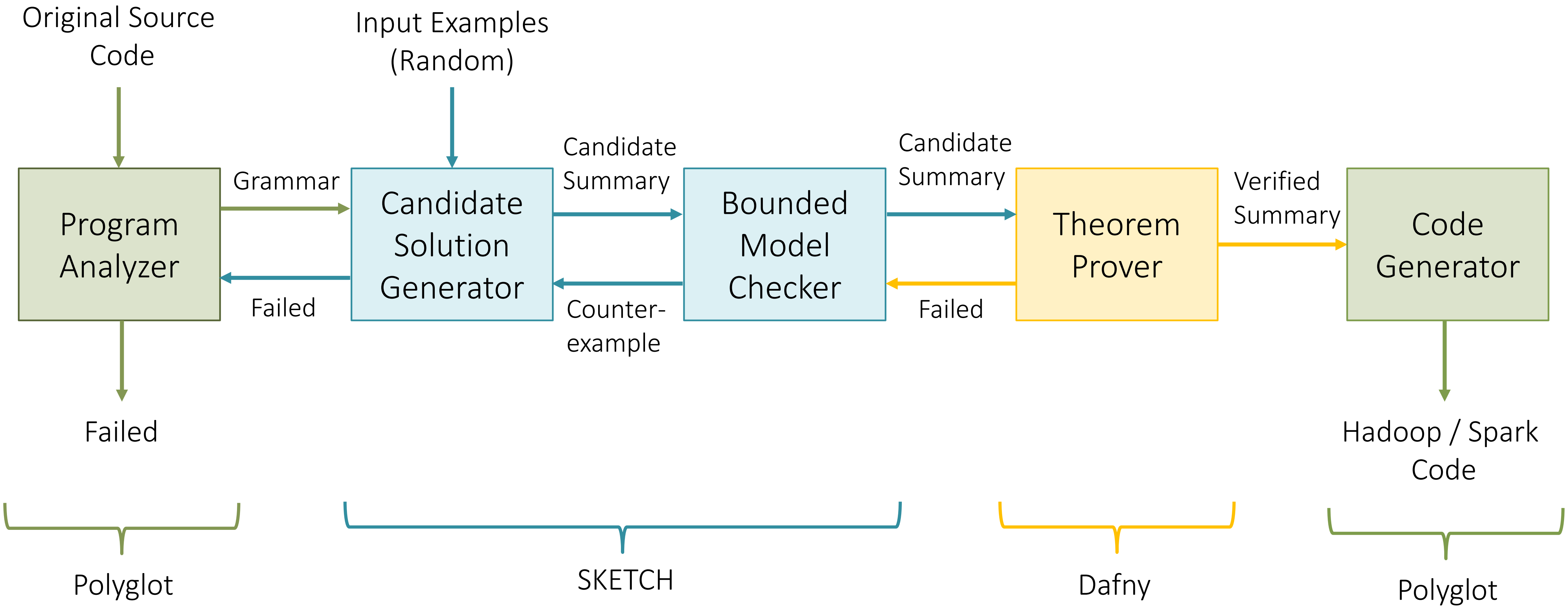
CASPER Architecture Diagram



CASPER Architecture Diagram



CASPER Architecture Diagram



Evaluation

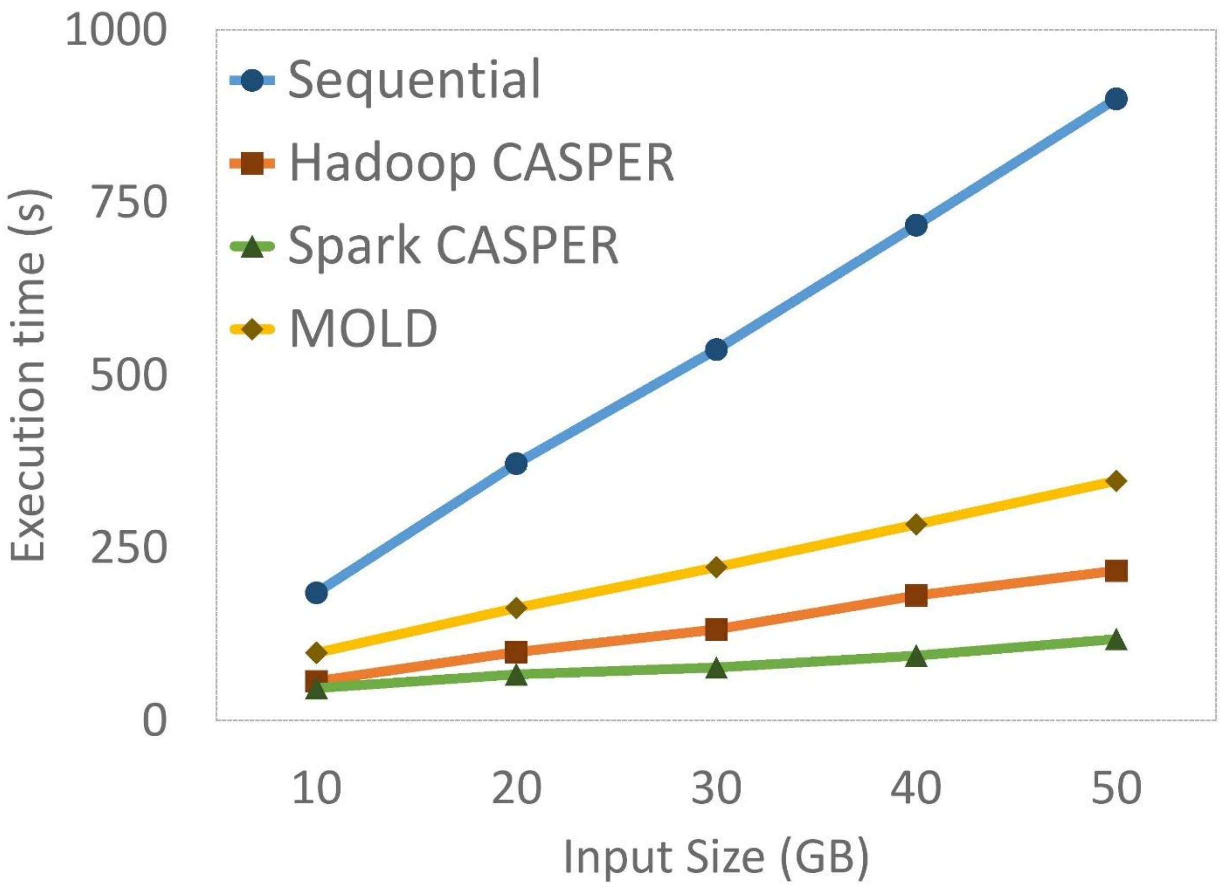
- Compilation performance
- Run-time performance
- Five benchmarks:
  - Summation
  - Word Count
  - String Search (Grep)
  - Linear Regression
  - 3D Histogram

Compilation Performance

Benchmark	Program Analysis	Synthesis and BMC	# of grammar Iterations	Formal Verification
Summation	< 1s	13s	1	2.8s
Word Count	< 1s	44s	1	3.4s
String Match	< 1s	1406s	2	3.3s
3D Histogram	< 1s	2355s	2	4.2s
Linear Regression	< 1s	1801s	2	4.8s

Runtime Performance

- Configuration:-
  - 10 node cluster
  - 8 vCPU, 15GB Memory
  - HDFS for data storage
  - Hadoop 2.7.2 and Spark 1.6.1
- Average Speedup:
  - **6.1x** on Spark
  - **3.3x** on Hadoop

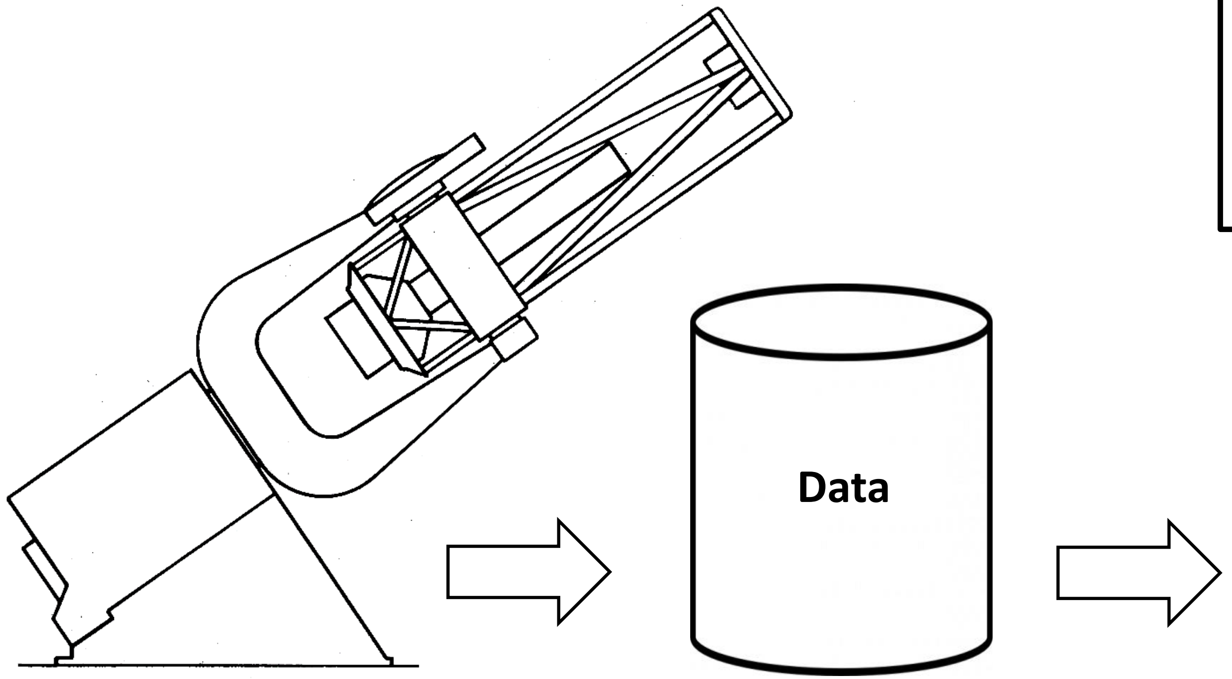


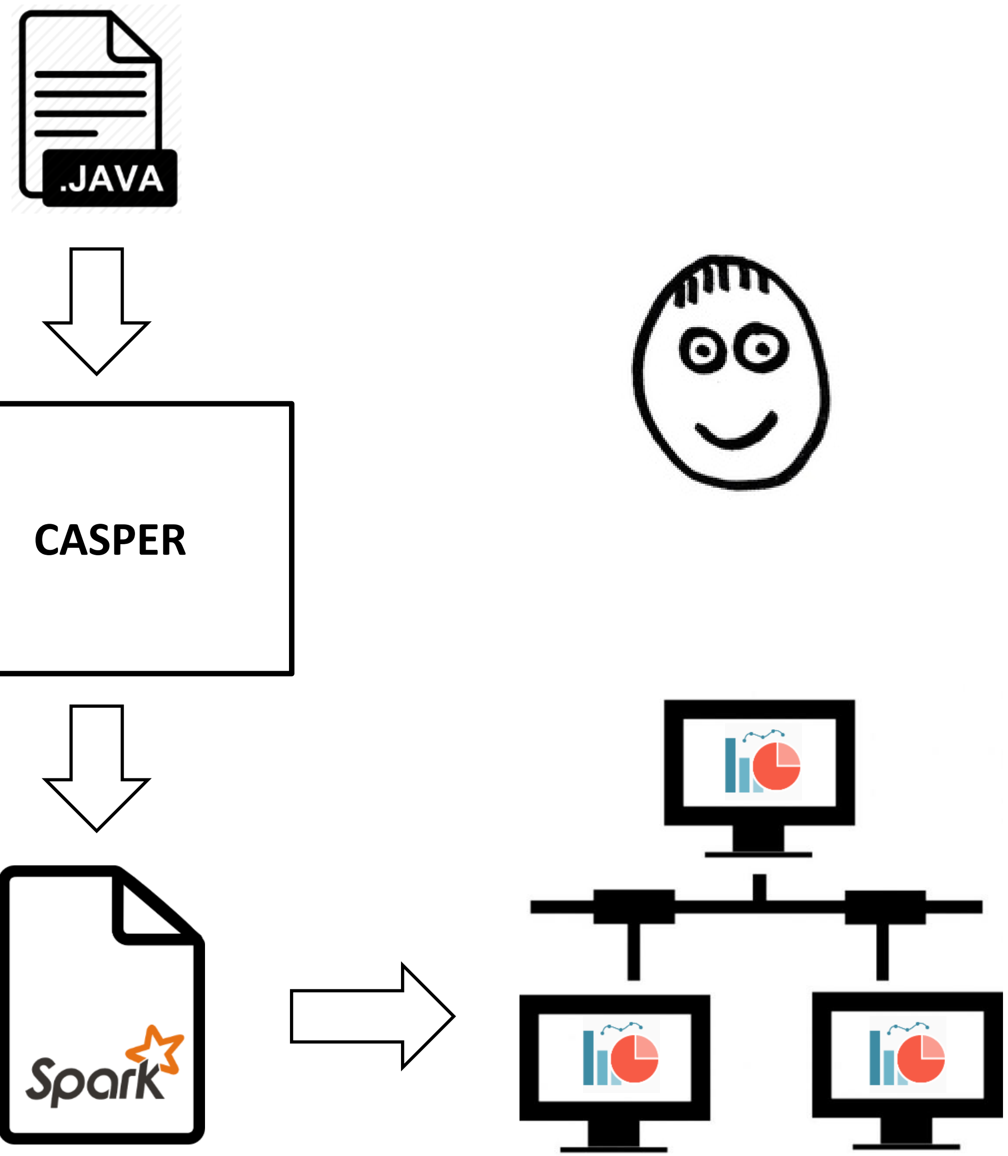
Benchmark: String Matching (Grep)

Demo!

Summary

Web-page: http://tinyurl.com/casper-homepage
Mailing-list: http://tinyurl.com/casper-subscribe





Data Collection Tool

Data Analytics Application
(Spark)