SANER'17

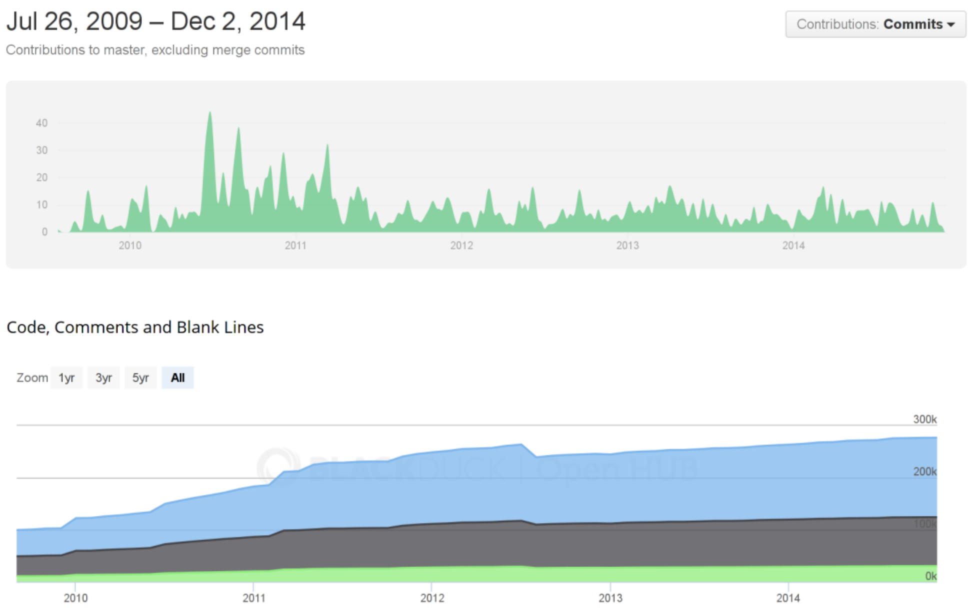
Klagenfurt, Austria

Reducing Redundancies in Multi-Revision Code Analysis

Carol V. Alexandru, Sebastiano Panichella, Harald C. Gall
Software Evolution and Architecture Lab
University of Zurich, Switzerland
{alexandru,panichella,gall}@ifi.uzh.ch
22.02.2017

The Problem Domain

- Static analysis (e.g. #Attr., McCabe, coupling...)



Jul 26, 2009 – Dec 2, 2014

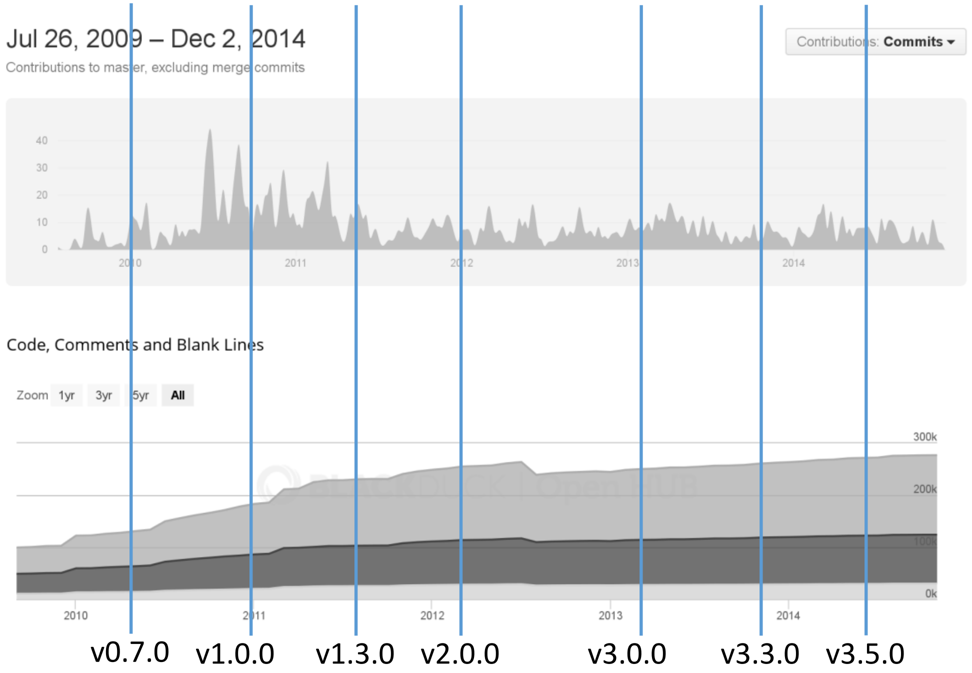
Contributions to master, excluding merge commits

Contributions: Commits ▾

Code, Comments and Blank Lines

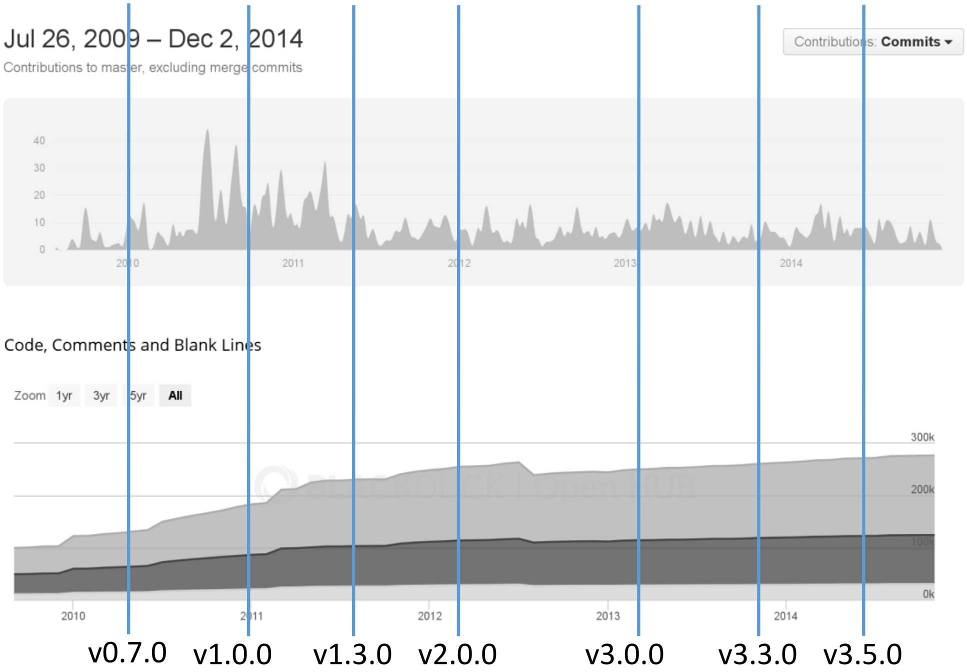
The Problem Domain

- Static analysis (e.g. #Attr., McCabe, coupling...)

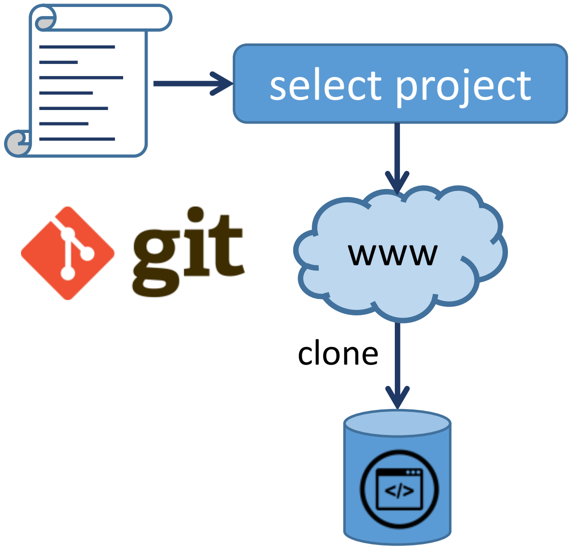


The Problem Domain

- Static analysis (e.g. #Attr., McCabe, coupling...)
- Many revisions, fine-grained historical data



A Typical Analysis Process

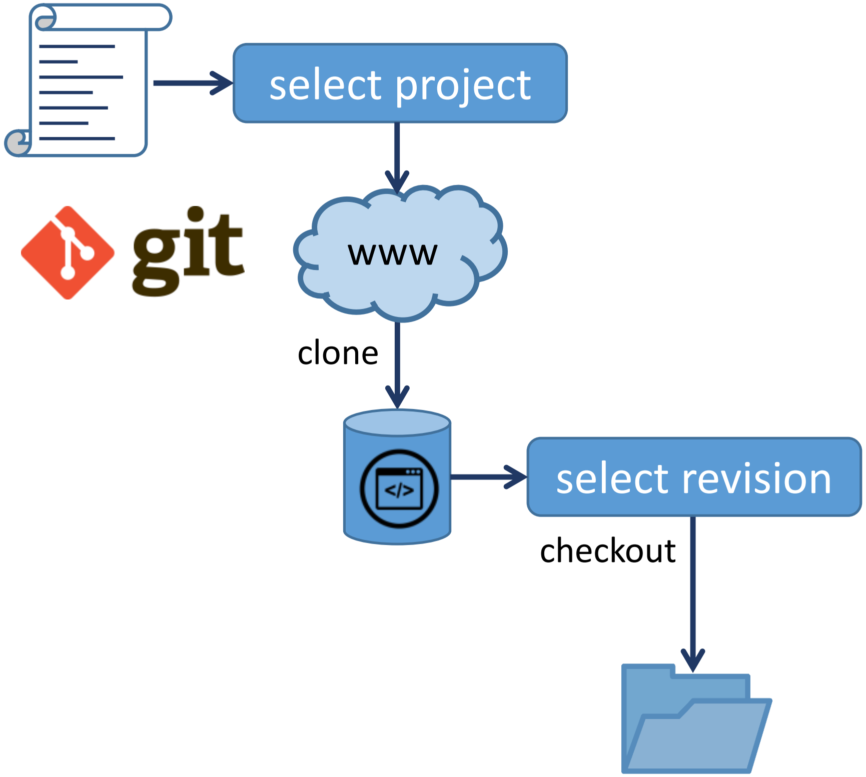


select project

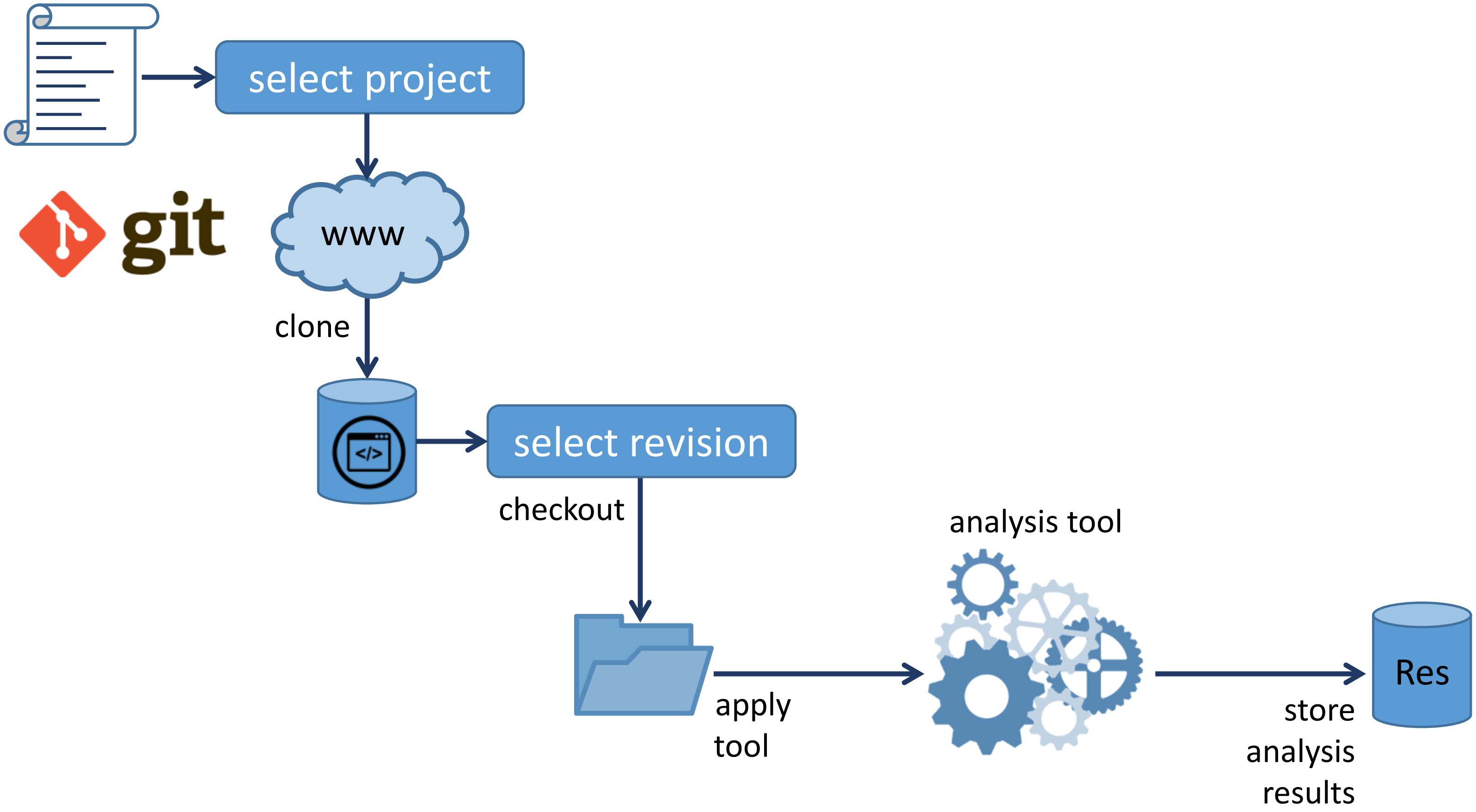
www

clone

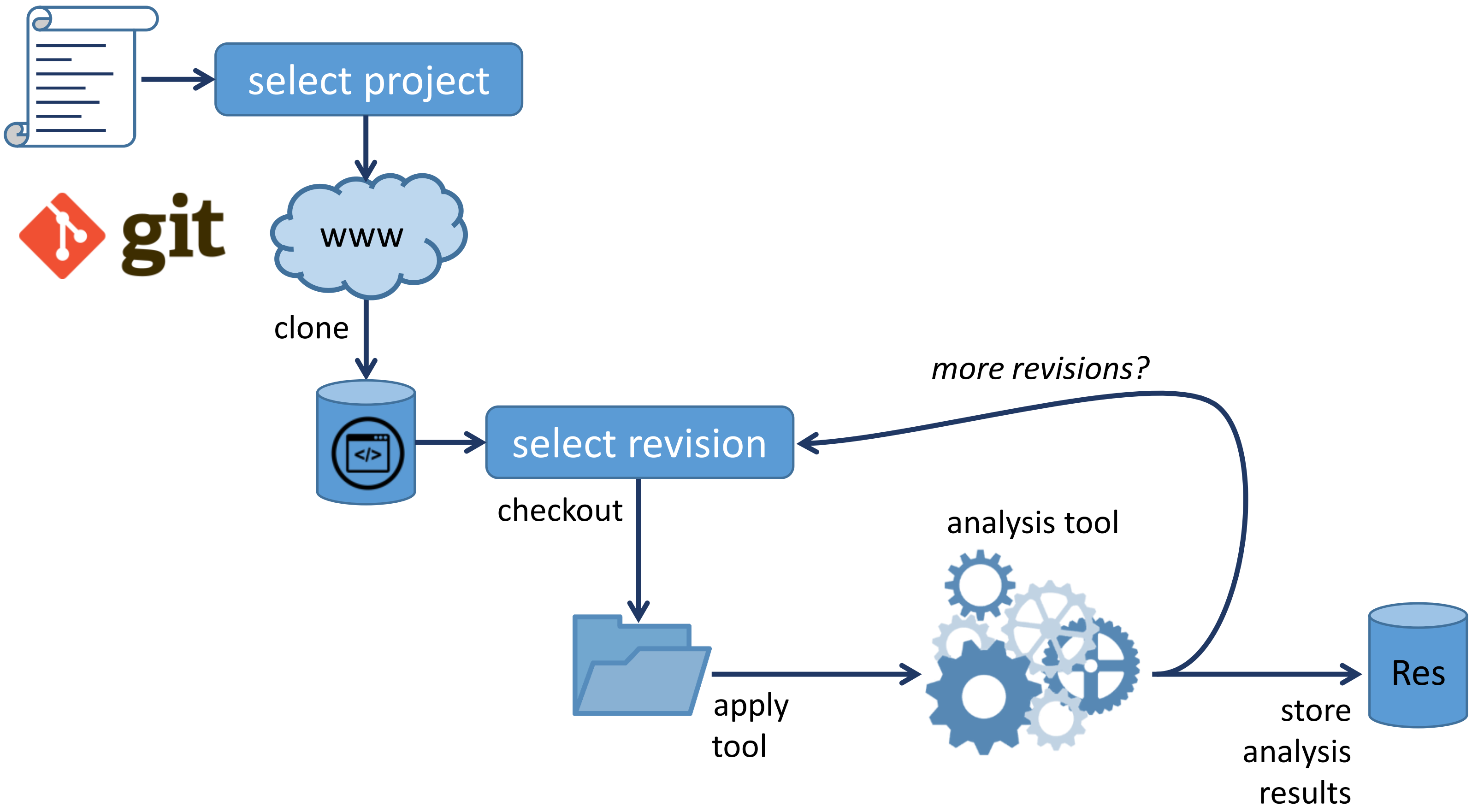
A Typical Analysis Process



A Typical Analysis Process



A Typical Analysis Process



select project

www

clone

select revision

checkout

more revisions?

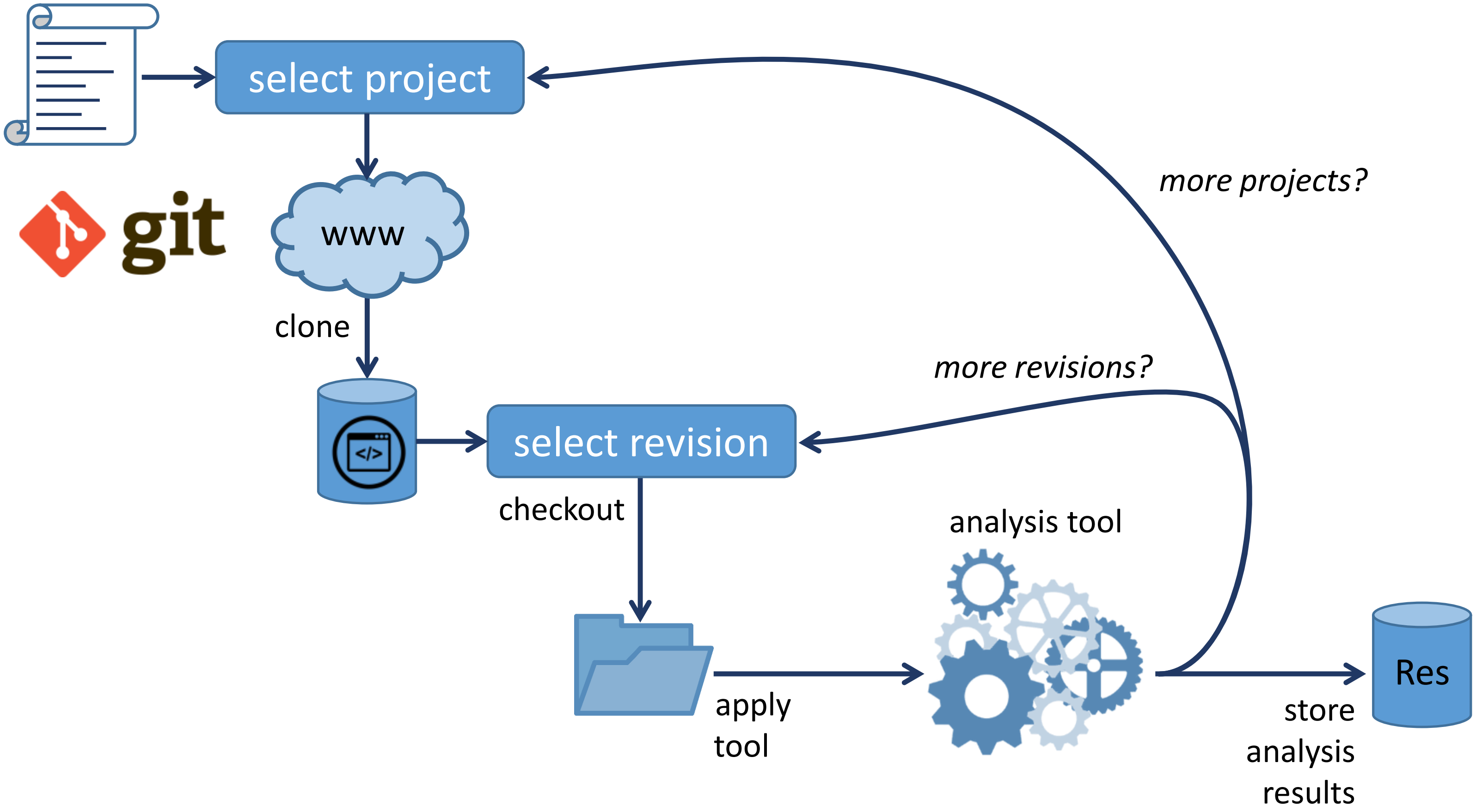
analysis tool

apply tool

Res

store analysis results

A Typical Analysis Process



select project

www

clone

select revision

checkout

apply tool

analysis tool

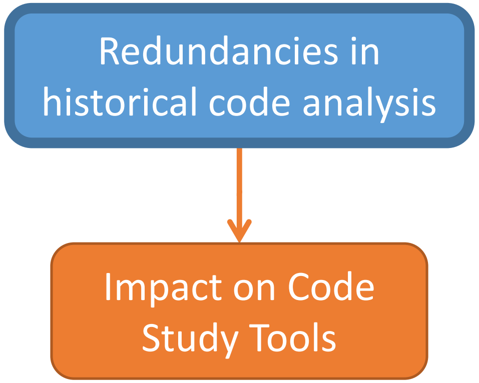
store analysis results

Res

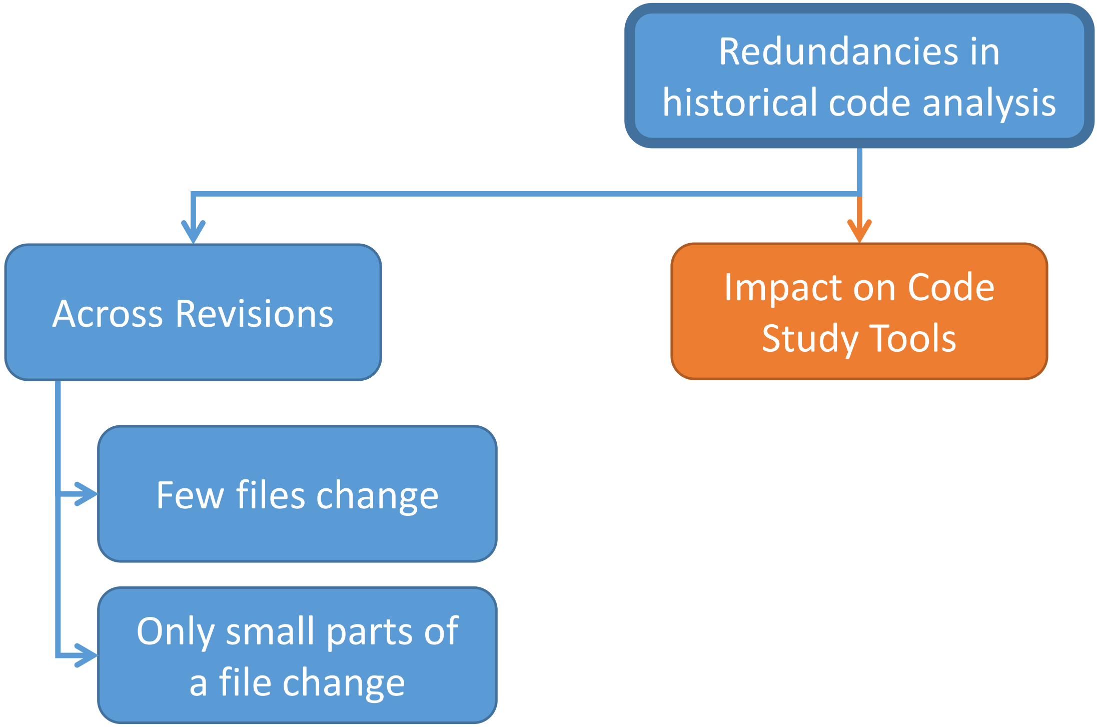
more revisions?

more projects?

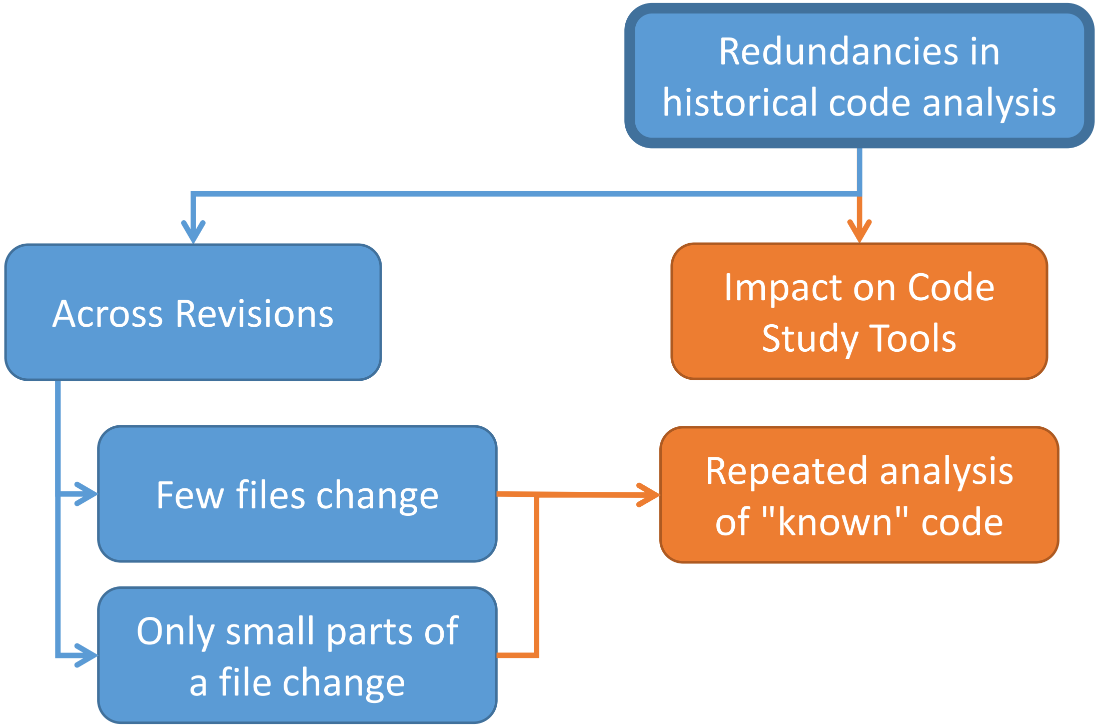
Redundancies all over…


Redundancies in historical code analysis
Impact on Code Study Tools

Redundancies all over...


Redundancies in historical code analysis
Across Revisions
Impact on Code Study Tools
Few files change
Only small parts of a file change

Redundancies all over...



Redundancies in historical code analysis

Across Revisions

Impact on Code Study Tools

Few files change

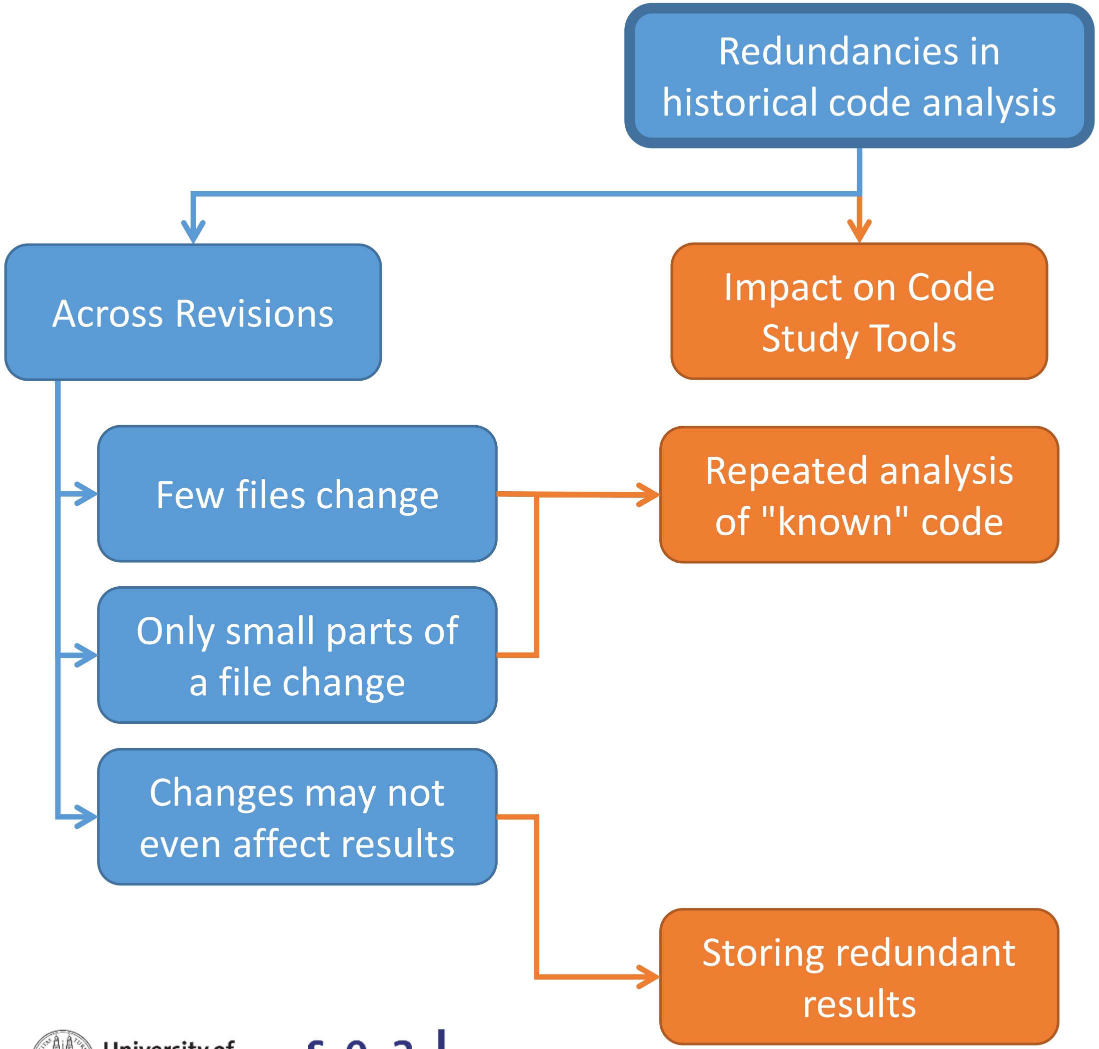
Only small parts of a file change

Repeated analysis of "known" code

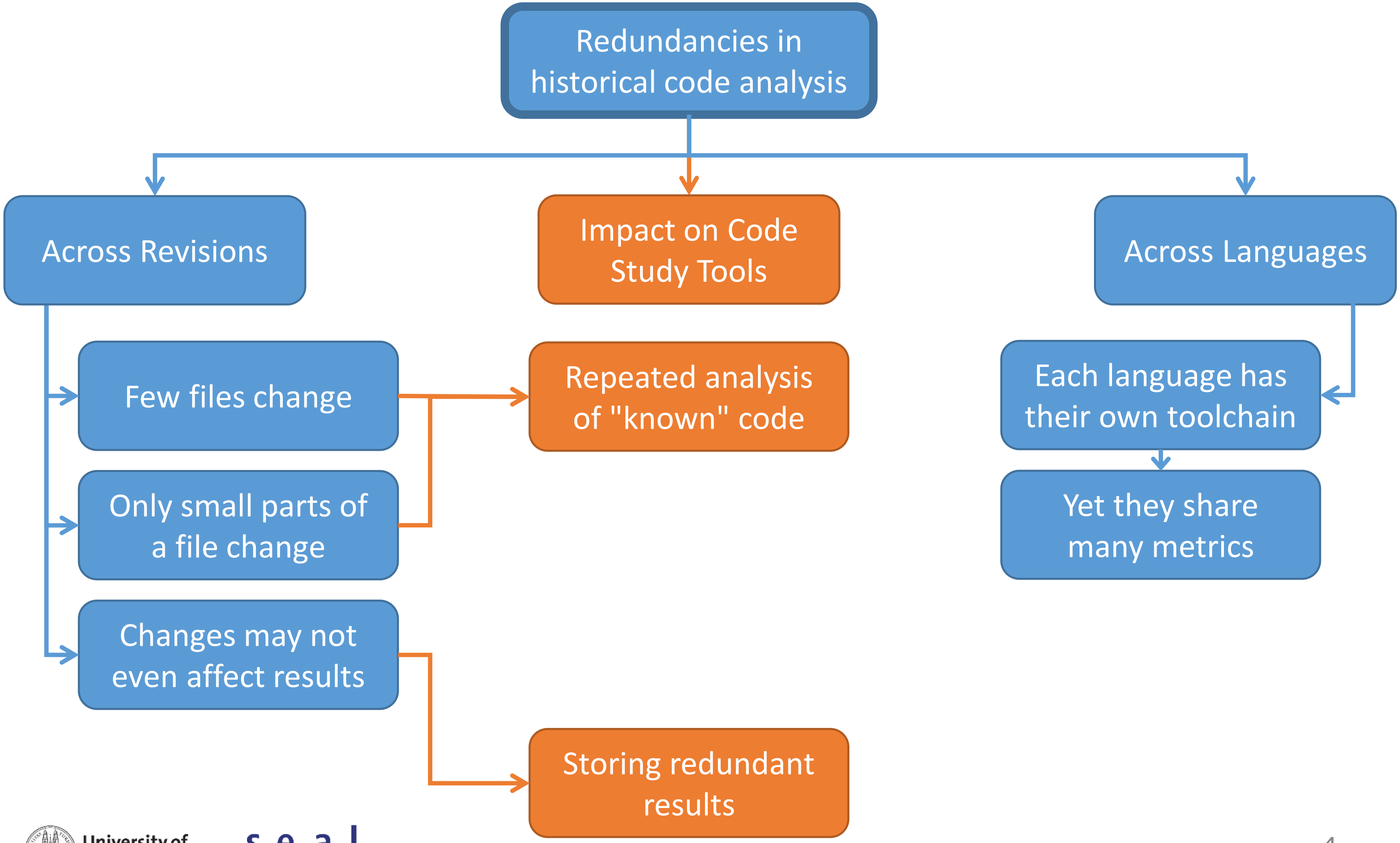
University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Redundancies all over...


Redundancies in historical code analysis
Across Revisions
Impact on Code Study Tools
Few files change
Only small parts of a file change
Changes may not even affect results
Repeated analysis of "known" code
Storing redundant results

Redundancies all over…



Redundancies in historical code analysis

Across Revisions

Impact on Code Study Tools

Across Languages

Few files change

Only small parts of a file change

Changes may not even affect results

Repeated analysis of "known" code

Storing redundant results

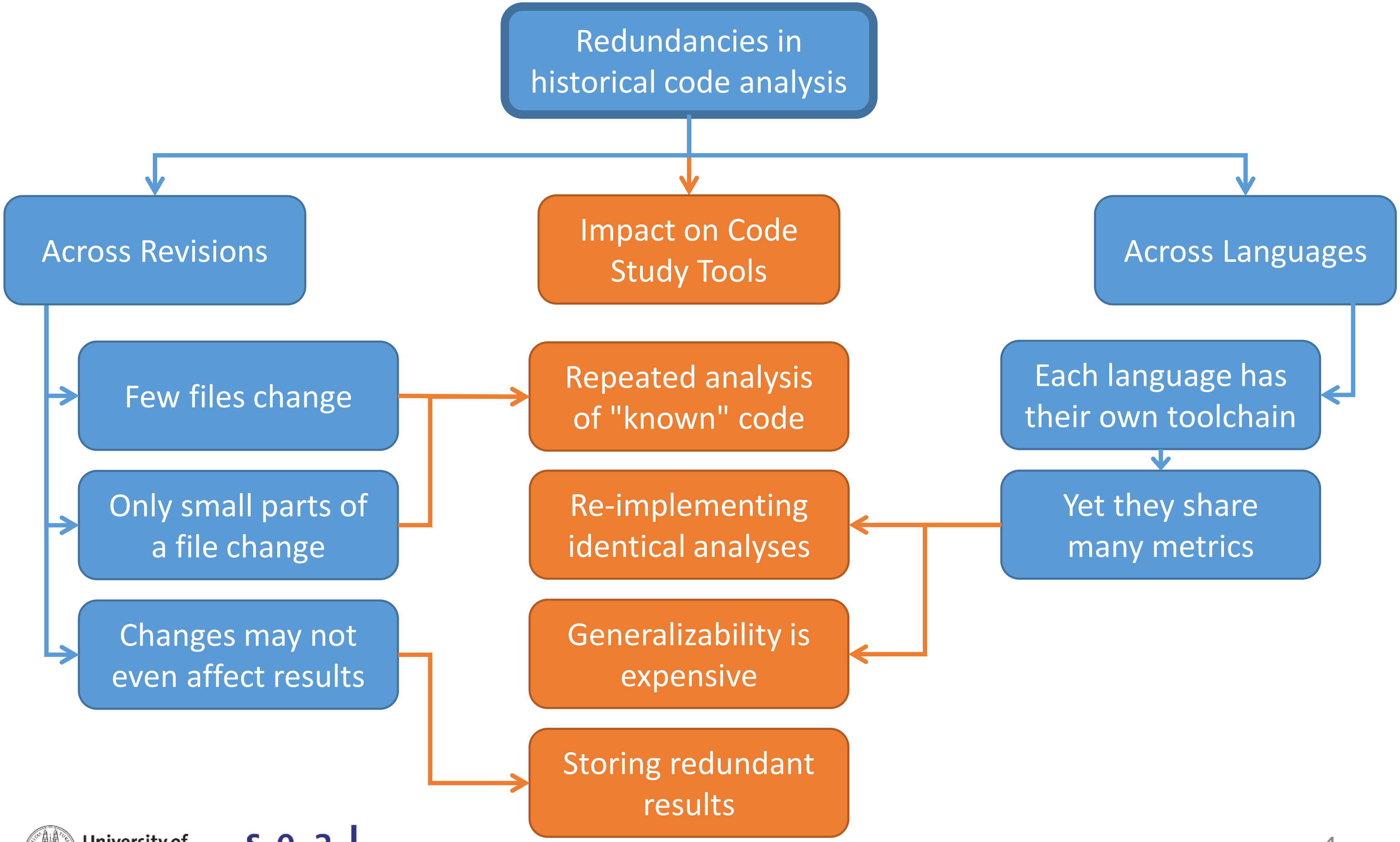
Each language has their own toolchain

Yet they share many metrics

University of Zurich^{UZH}

s.e.a.l. software evolution & architecture lab

Redundancies all over…



Redundancies in historical code analysis

Across Revisions

Few files change

Only small parts of a file change

Changes may not even affect results

Impact on Code Study Tools

Repeated analysis of "known" code

Re-implementing identical analyses

Generalizability is expensive

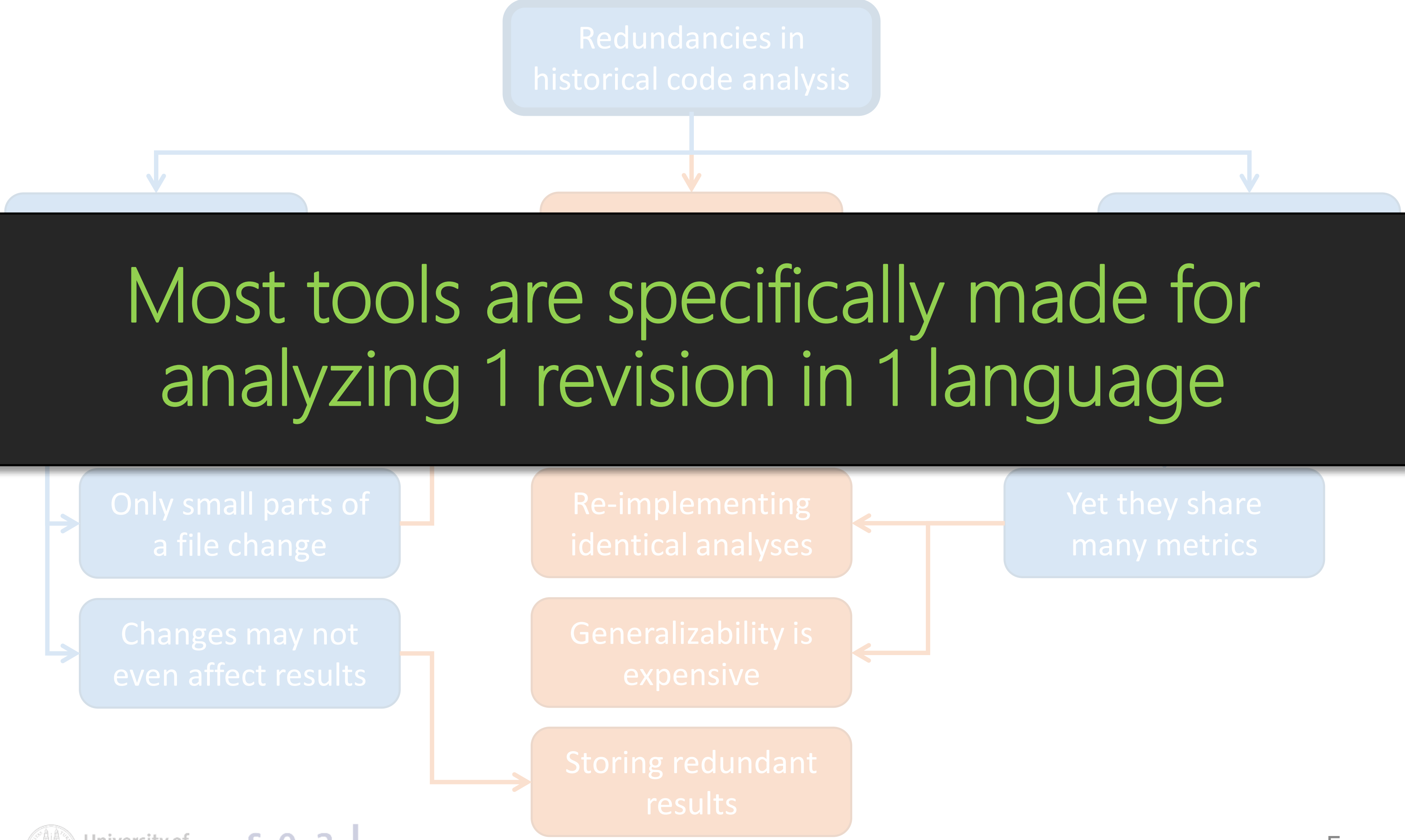
Storing redundant results

Across Languages

Each language has their own toolchain

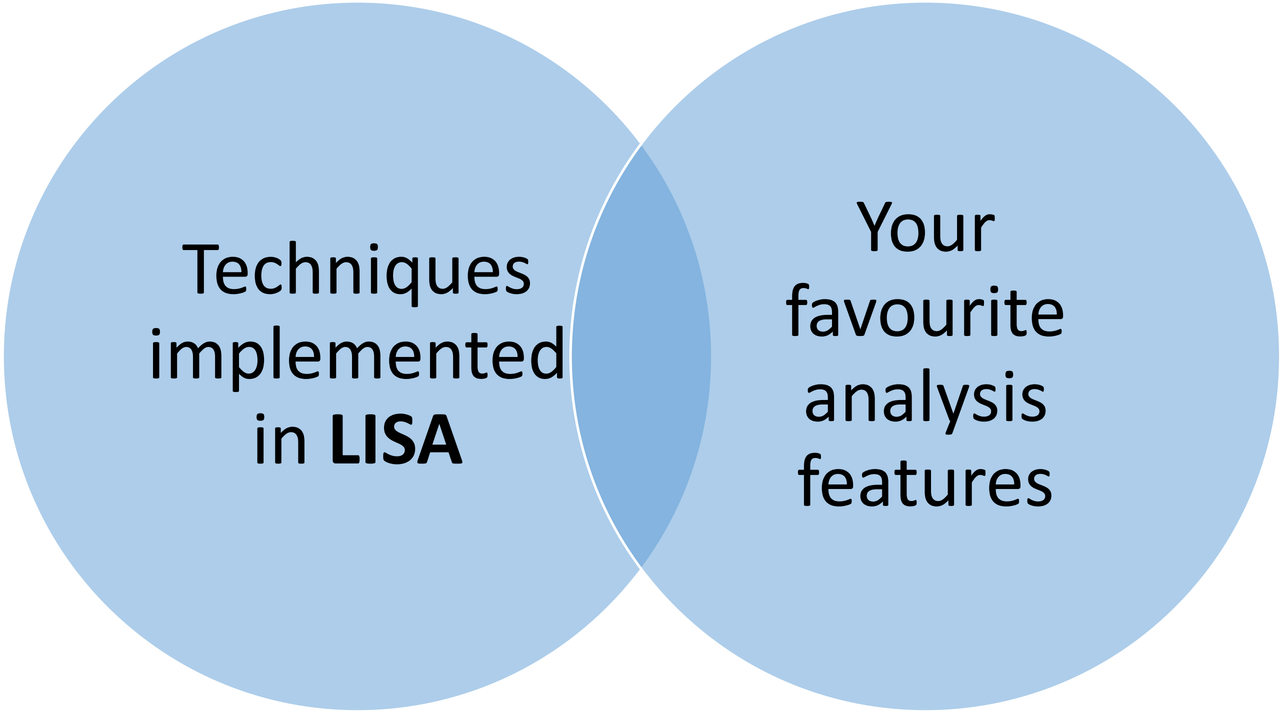
Yet they share many metrics

Redundancies all over…


Redundancies in historical code analysis
Only small parts of a file change
Changes may not even affect results
Re-implementing identical analyses
Generalizability is expensive
Storing redundant results
Yet they share many metrics

Most tools are specifically made for analyzing 1 revision in 1 language

Important!



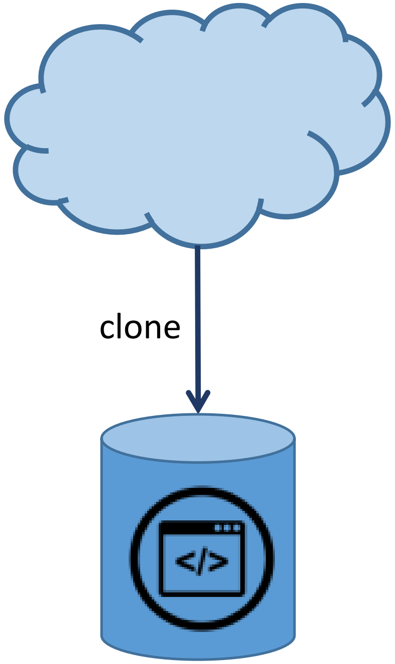
Techniques implemented in **LISA**

Your favourite analysis features

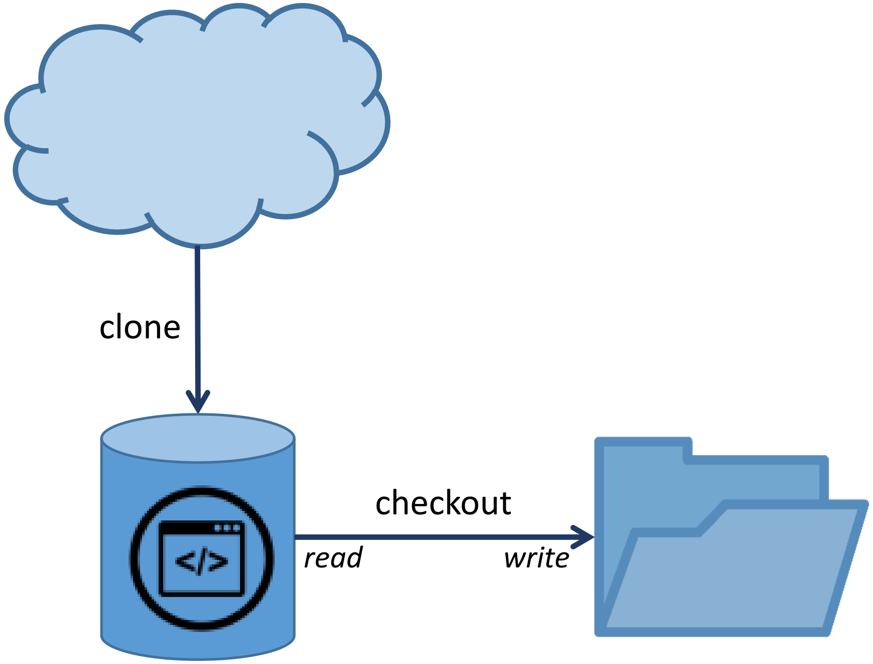
Pick what you like!

#1: Avoid Checkouts

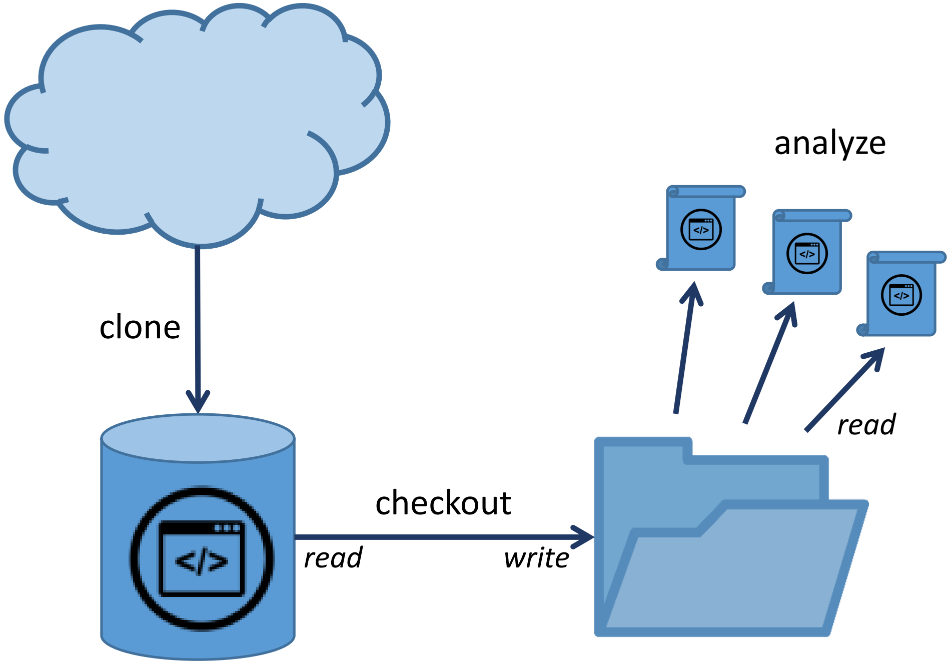
Avoid checkouts



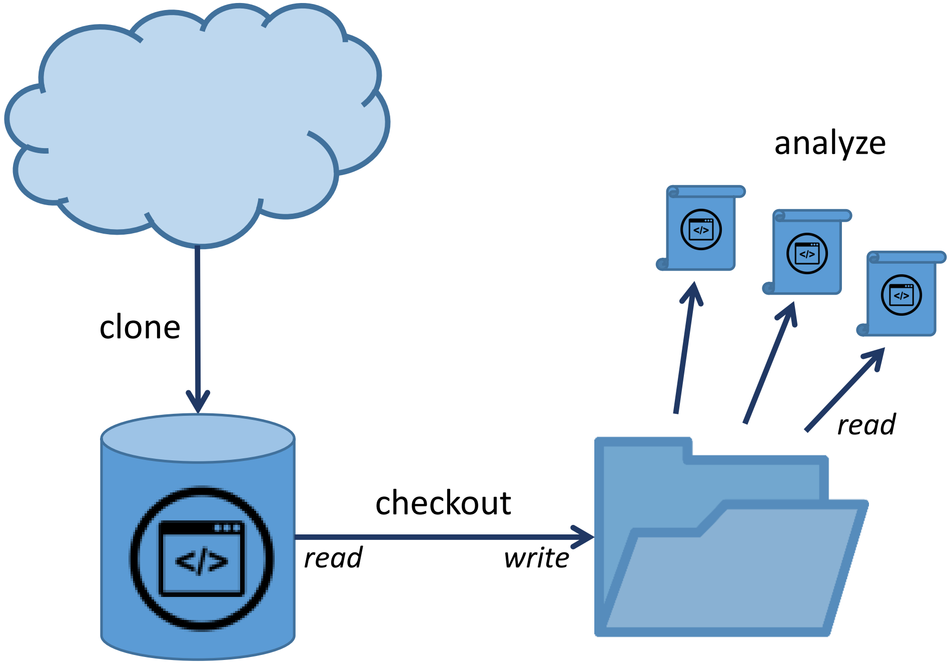
Avoid checkouts



Avoid checkouts

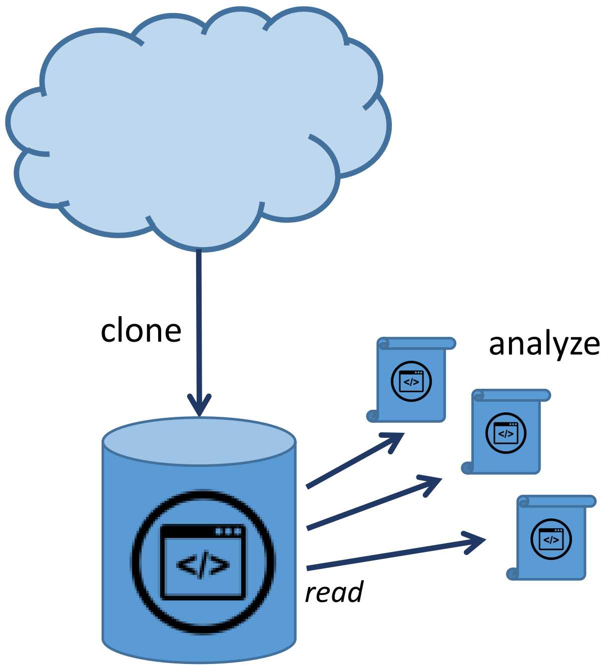


Avoid checkouts

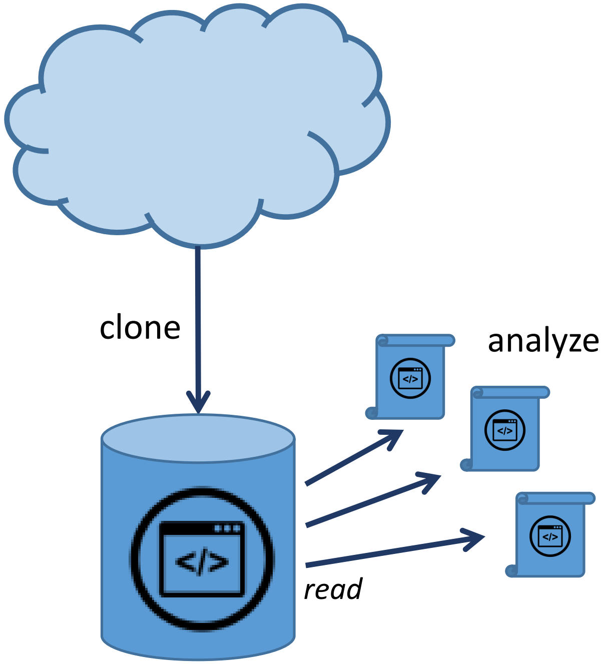


For every file: 2 read ops + 1 write op
Checkout includes irrelevant files
Need 1 CWD for every revision to be analyzed in parallel

Avoid checkouts

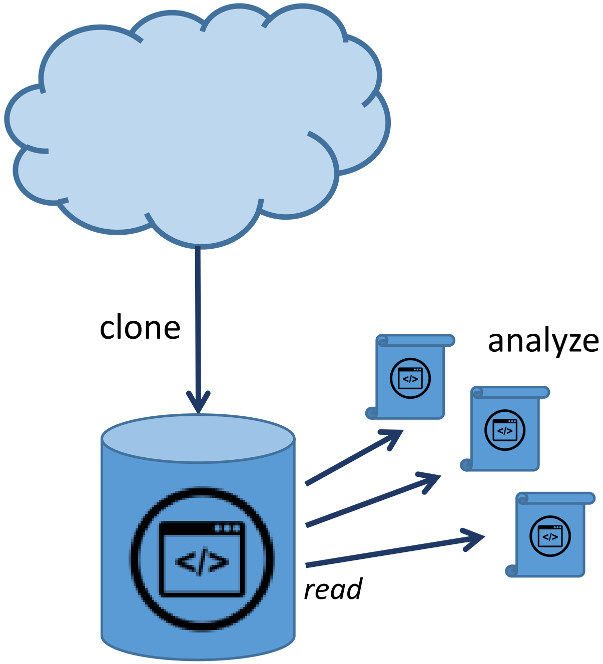


Avoid checkouts

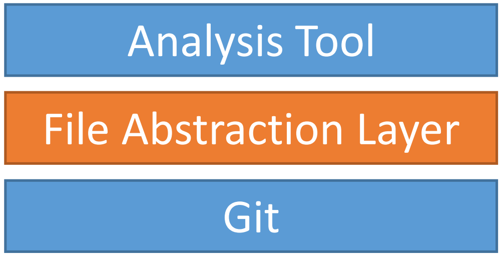


Only read relevant files in a single read op
No write ops
No overhead for parallization

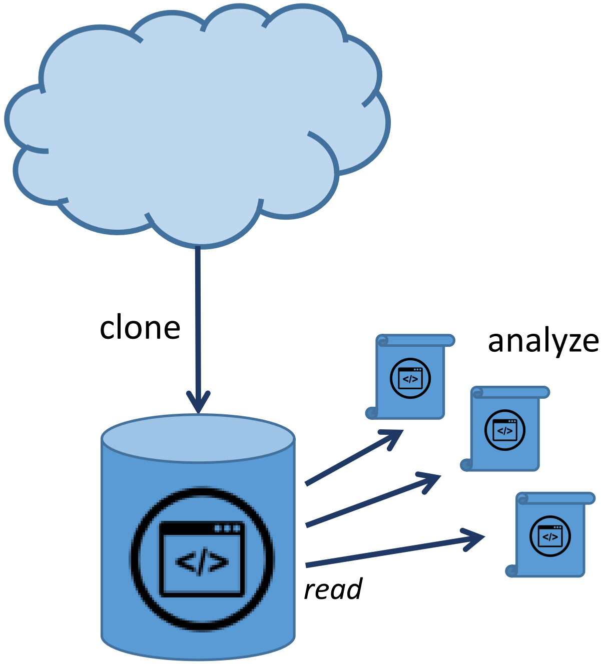
Avoid checkouts



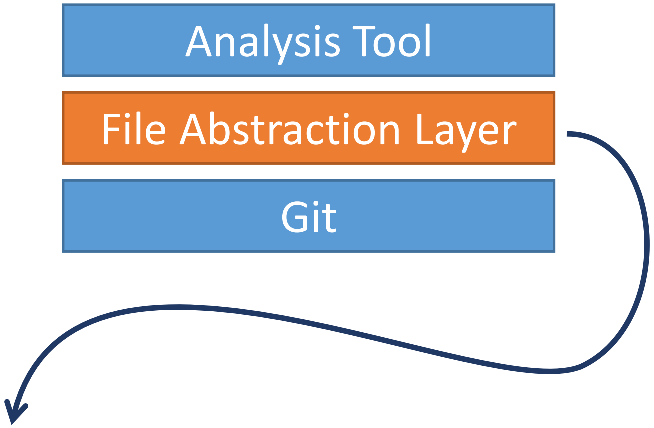
Only read relevant files in a single read op
No write ops
No overhead for parallization



Avoid checkouts



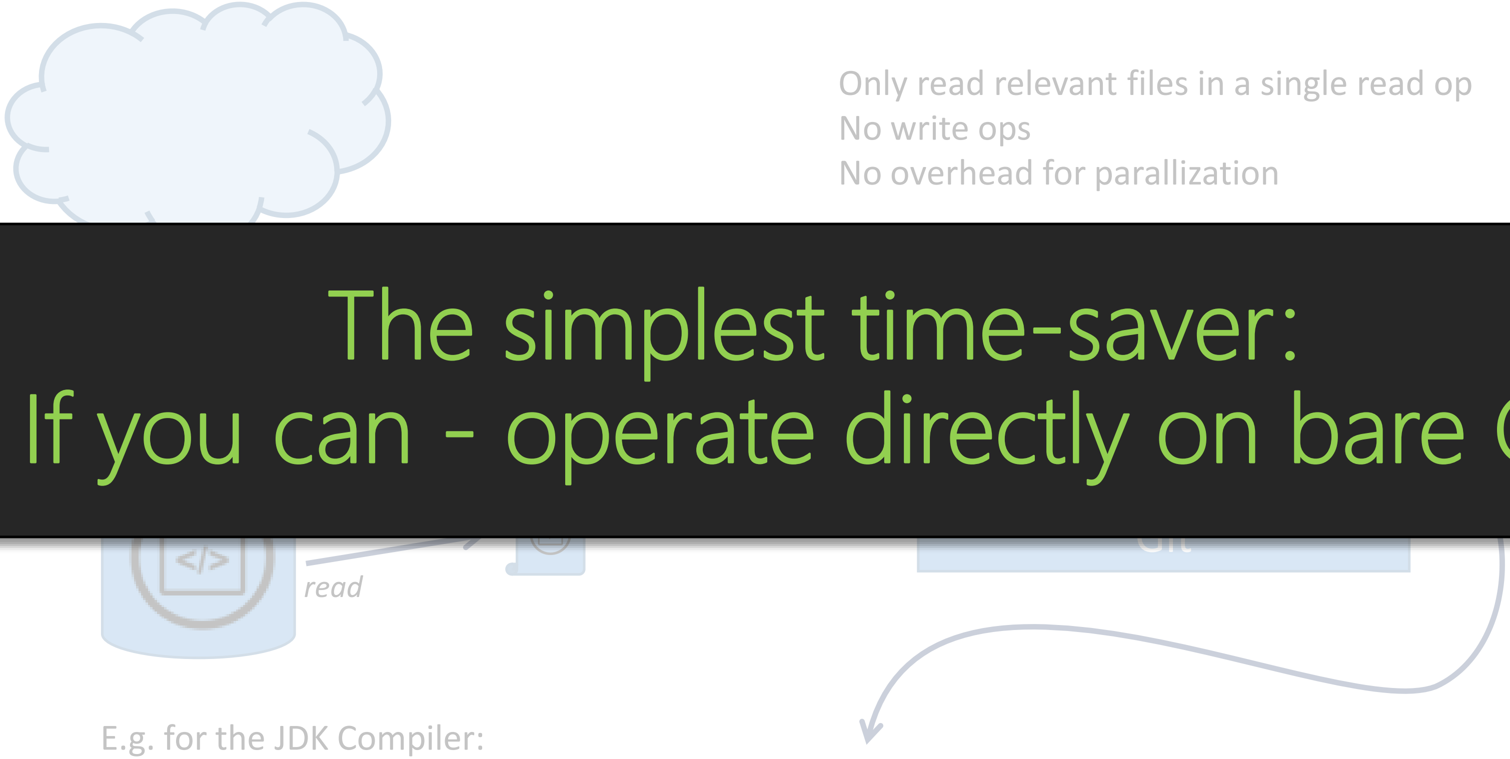
Only read relevant files in a single read op
No write ops
No overhead for parallization



E.g. for the JDK Compiler:

```
class JavaSourceFromCharrArray(name: String, val code: CharBuffer)
extends SimpleJavaFileObject(URI.create("string:///" + name), Kind.SOURCE) {
  override def getCharContent(): CharSequence = code
}
```

Avoid checkouts



Only read relevant files in a single read op
No write ops
No overhead for parallization

The simplest time-saver:
If you can - operate directly on bare Git

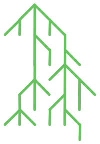
E.g. for the JDK Compiler:

```
class JavaSourceFromCharrArray(name: String, val code: CharBuffer)
extends SimpleJavaFileObject(URI.create("string:///" + name), Kind.SOURCE) {
  override def getCharContent(): CharSequence = code
}
```

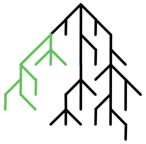
#2: Use a multi-revision representation of your sources

Merge ASTs

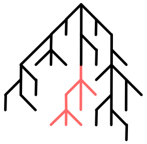
rev. 1



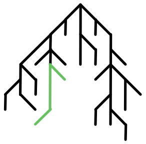
rev. 2



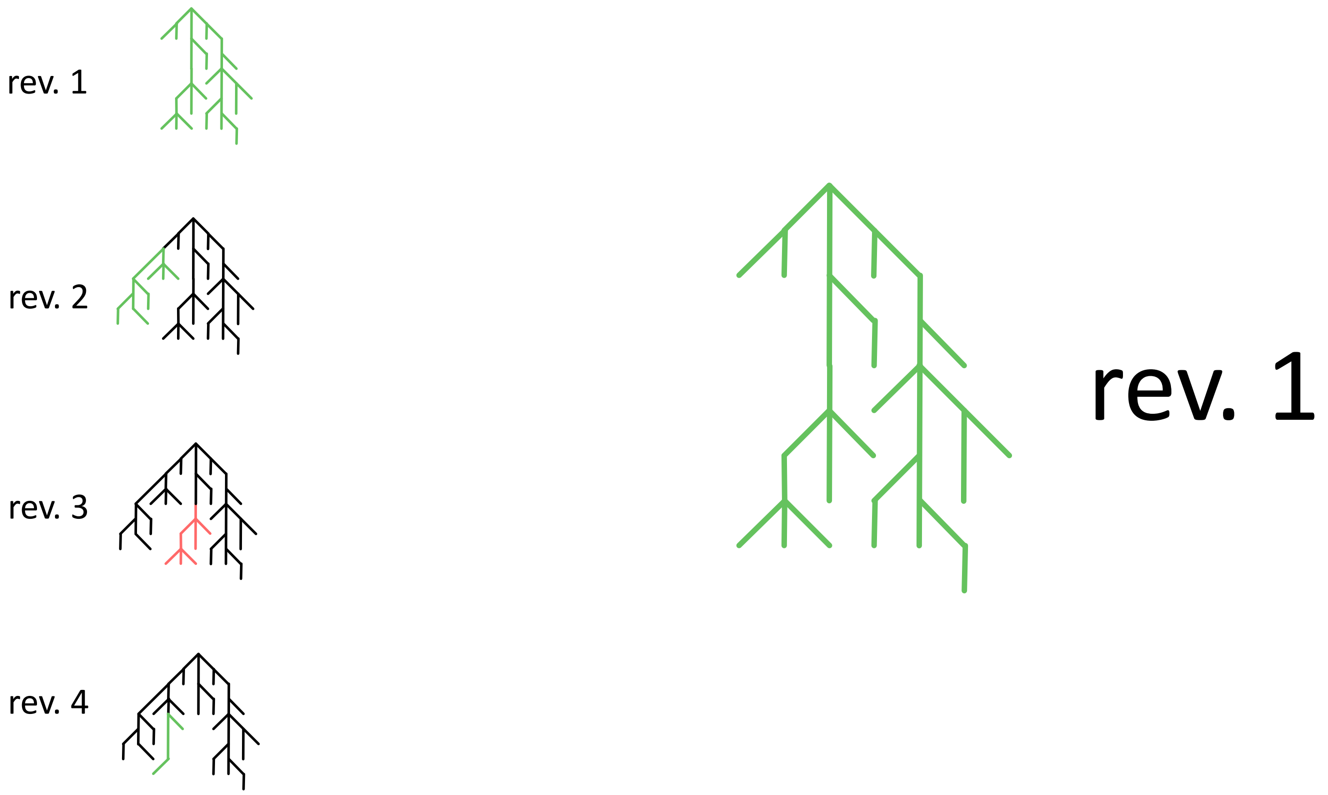
rev. 3



rev. 4



Merge ASTs



rev. 1

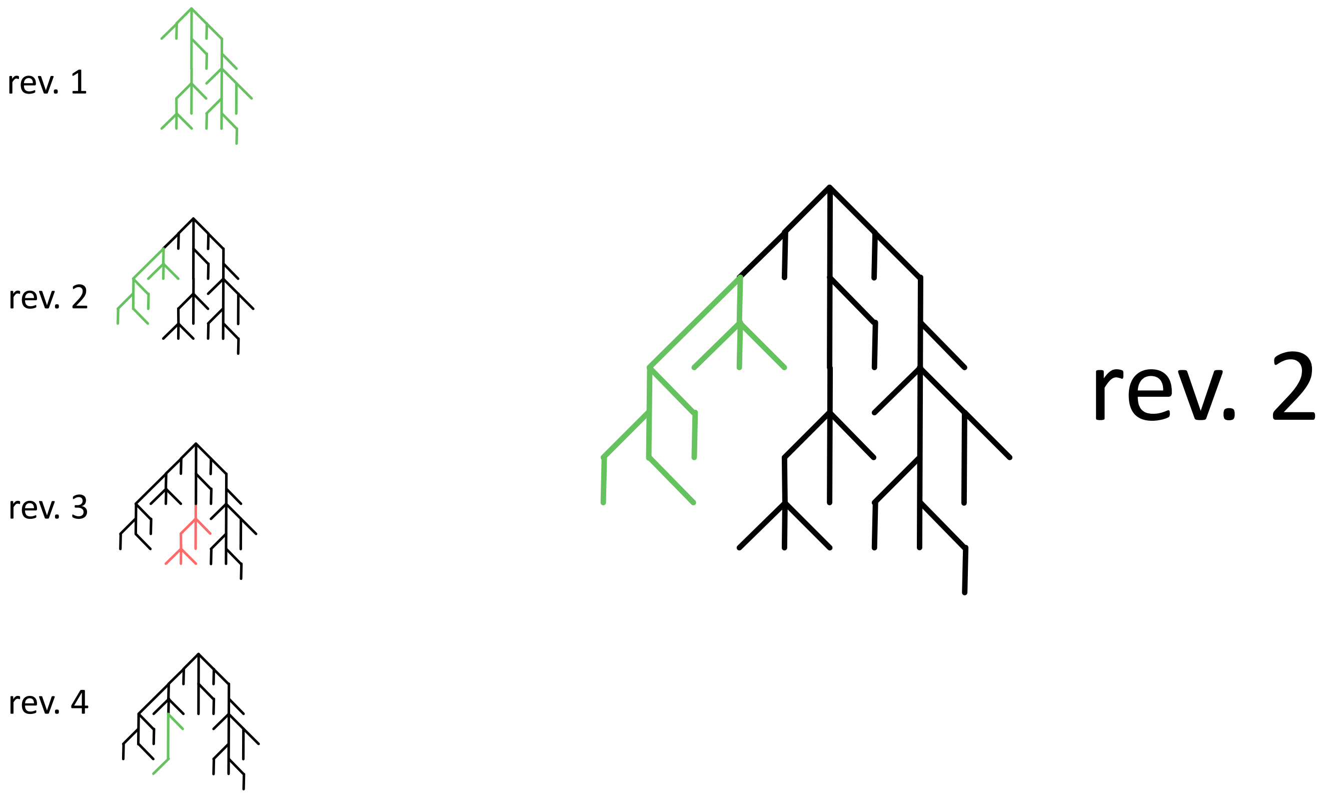
rev. 2

rev. 3

rev. 4

rev. 1

Merge ASTs



rev. 1

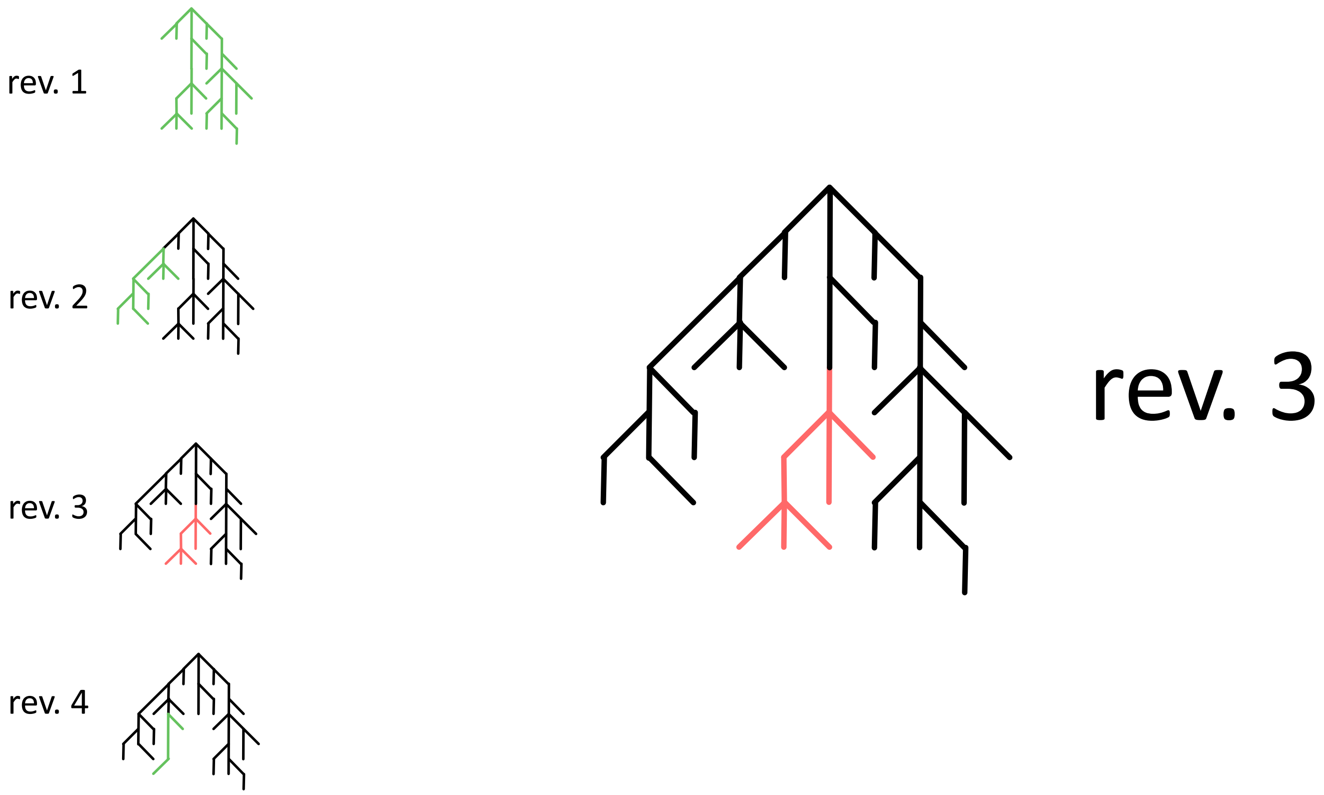
rev. 2

rev. 3

rev. 4

rev. 2

Merge ASTs



rev. 1

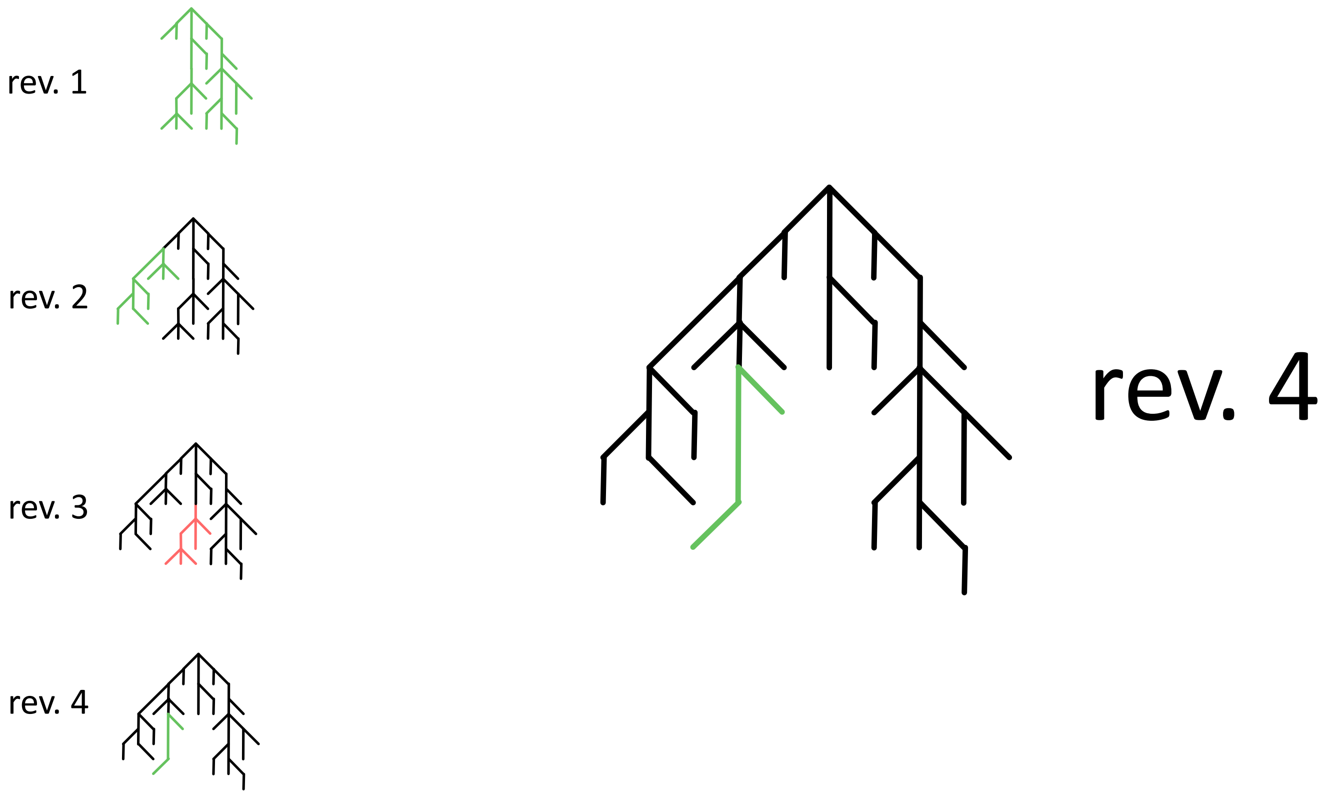
rev. 2

rev. 3

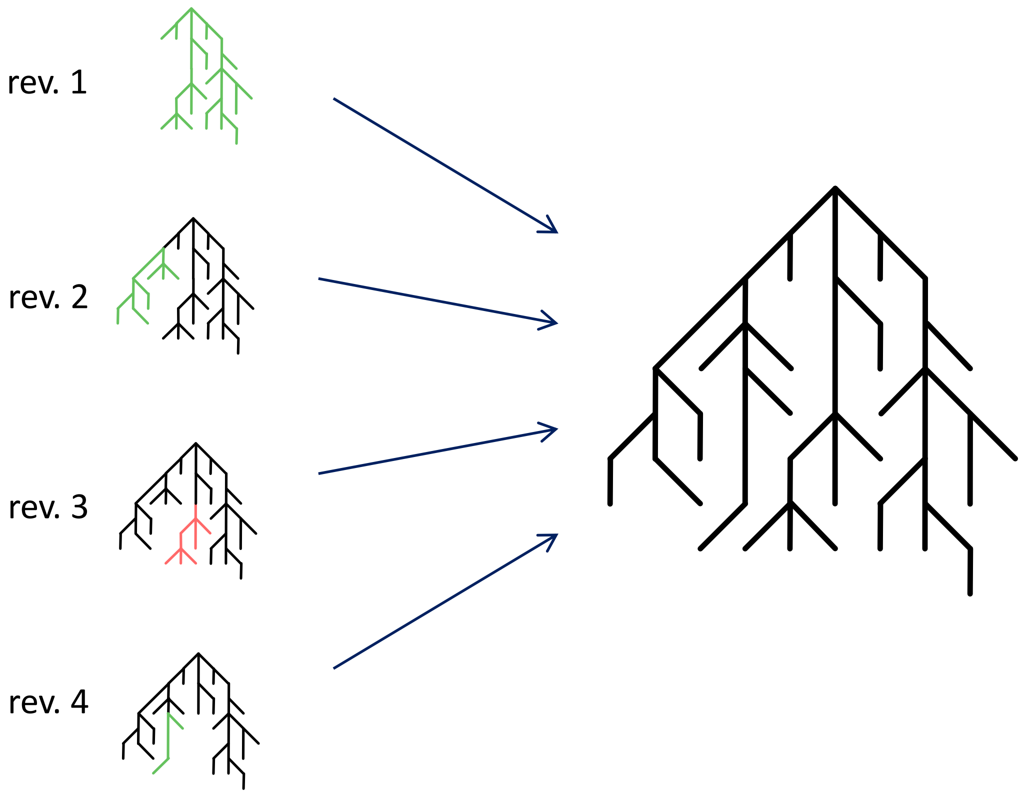
rev. 4

rev. 3

Merge ASTs



Merge ASTs



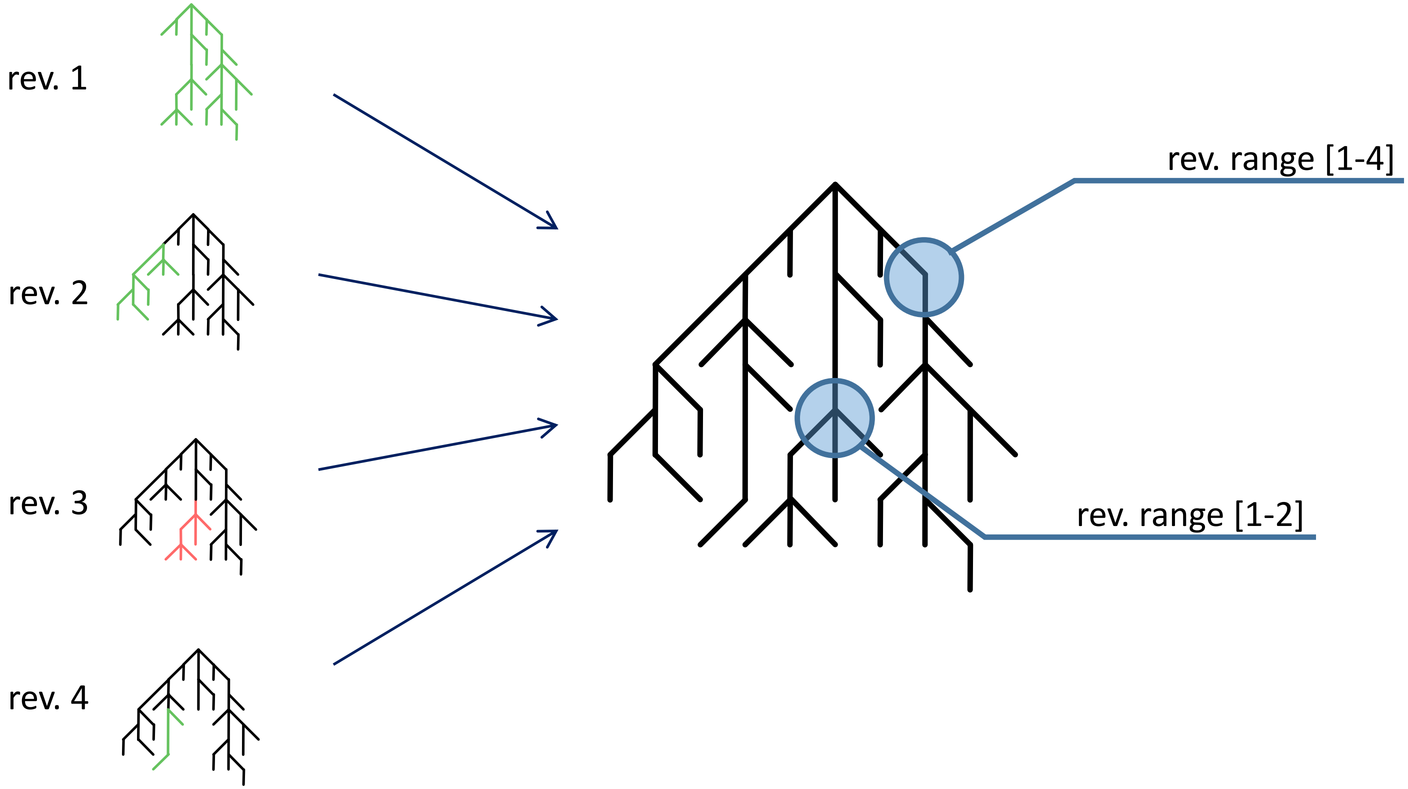
rev. 1

rev. 2

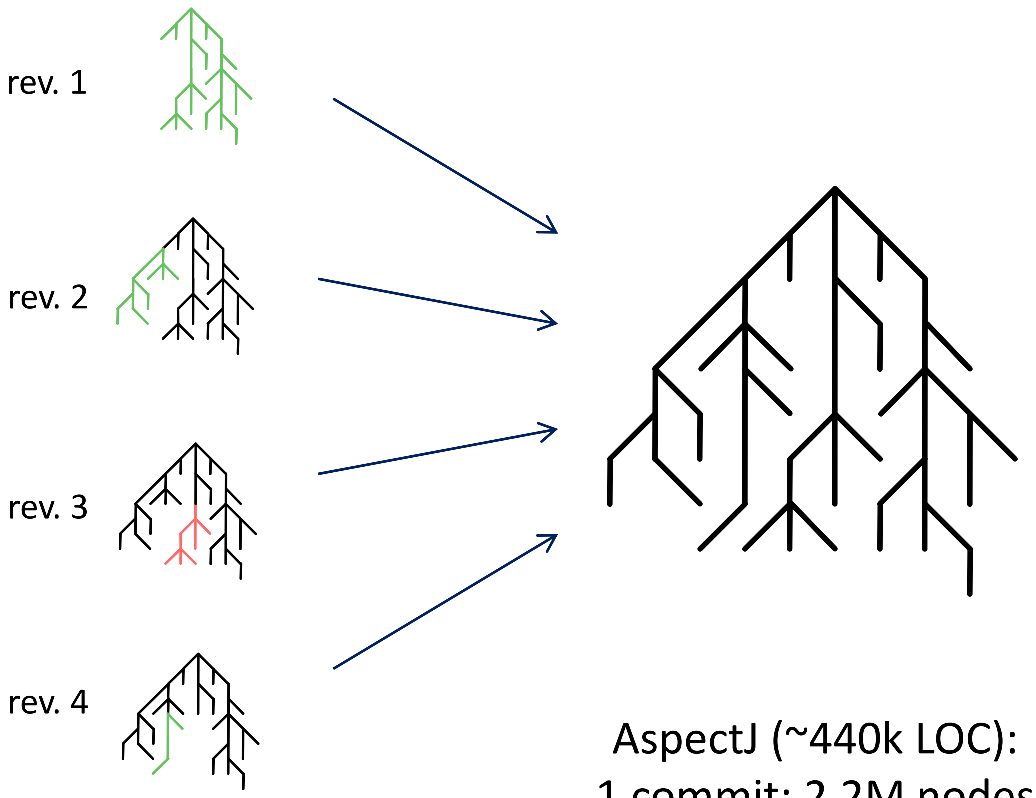
rev. 3

rev. 4

Merge ASTs



Merge ASTs



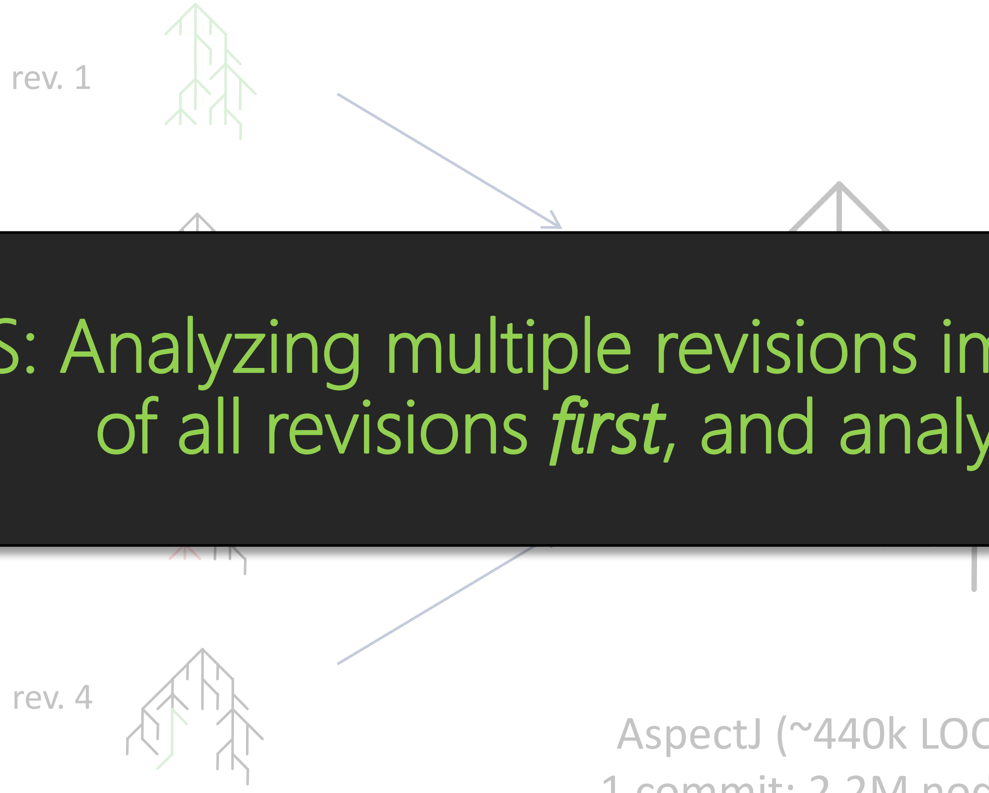
AspectJ (~440k LOC):
1 commit: 2.2M nodes
All >7000 commits: 6.5M nodes

Merge ASTs

Merging ASTs brings exponential space and time savings

AspectJ (~440k LOC):
1 commit: 2.2M nodes
All >7000 commits: 6.5M nodes

Merge ASTs



PS: Analyzing multiple revisions implies building a graph of all revisions *first*, and analyzing it *afterwards*

AspectJ (~440k LOC):
1 commit: 2.2M nodes
All >7000 commits: 6.5M nodes

#3: Store AST nodes only if they're needed for analysis

```
public class Demo {
  public void run() {
    for (int i = 1; i< 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
         System.out.println(i)
      }
  }
}
```

What's the complexity (1+#forks) and name for each method and class?

```
public class Demo {
  public void run() {
    for (int i = 1; i< 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
         System.out.println(i)
      }
  }
}
```

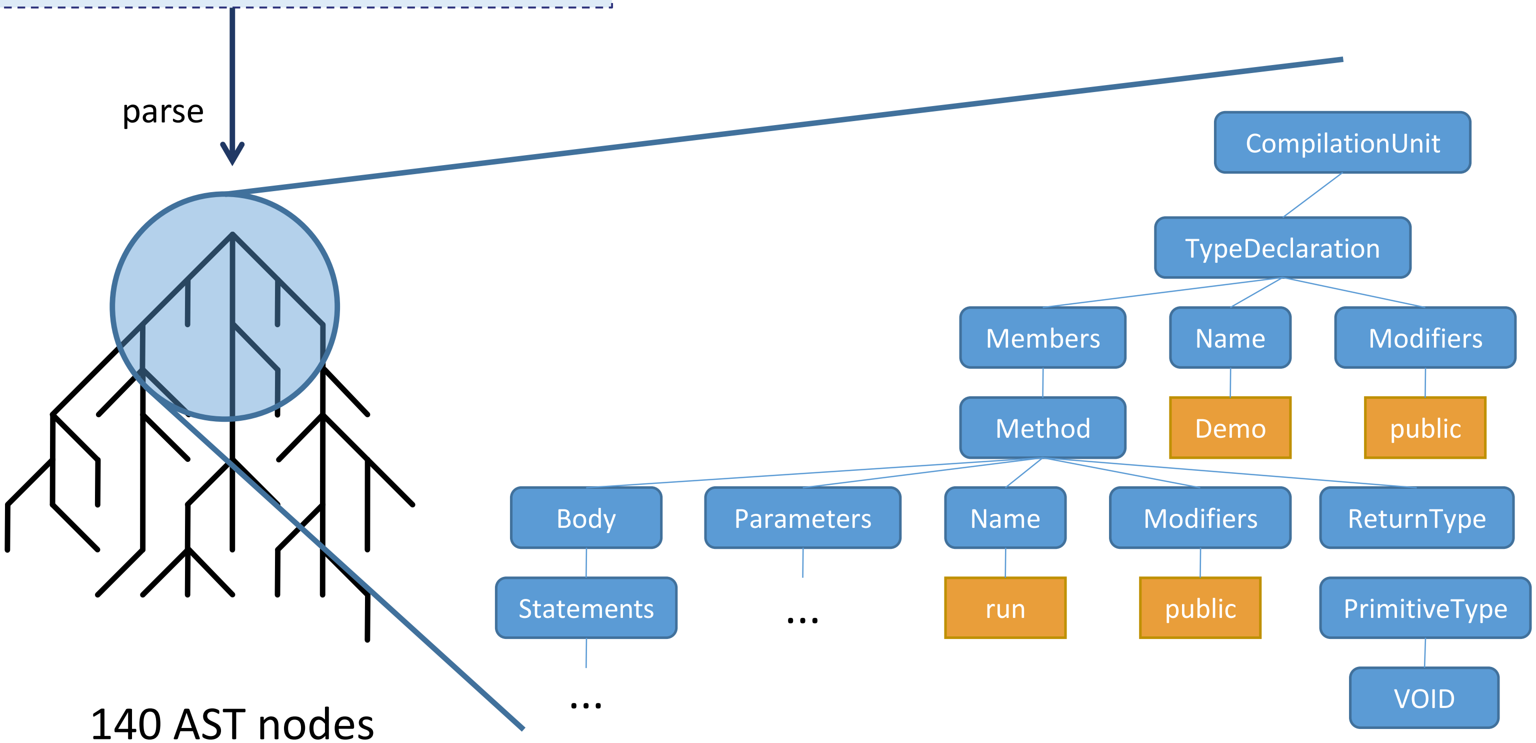
What's the complexity (1+#forks) and name for each method and class?

parse

140 AST nodes
(using ANTLR)

```
public class Demo {
  public void run() {
    for (int i = 1; i< 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
         System.out.println(i)
      }
  }
}
```

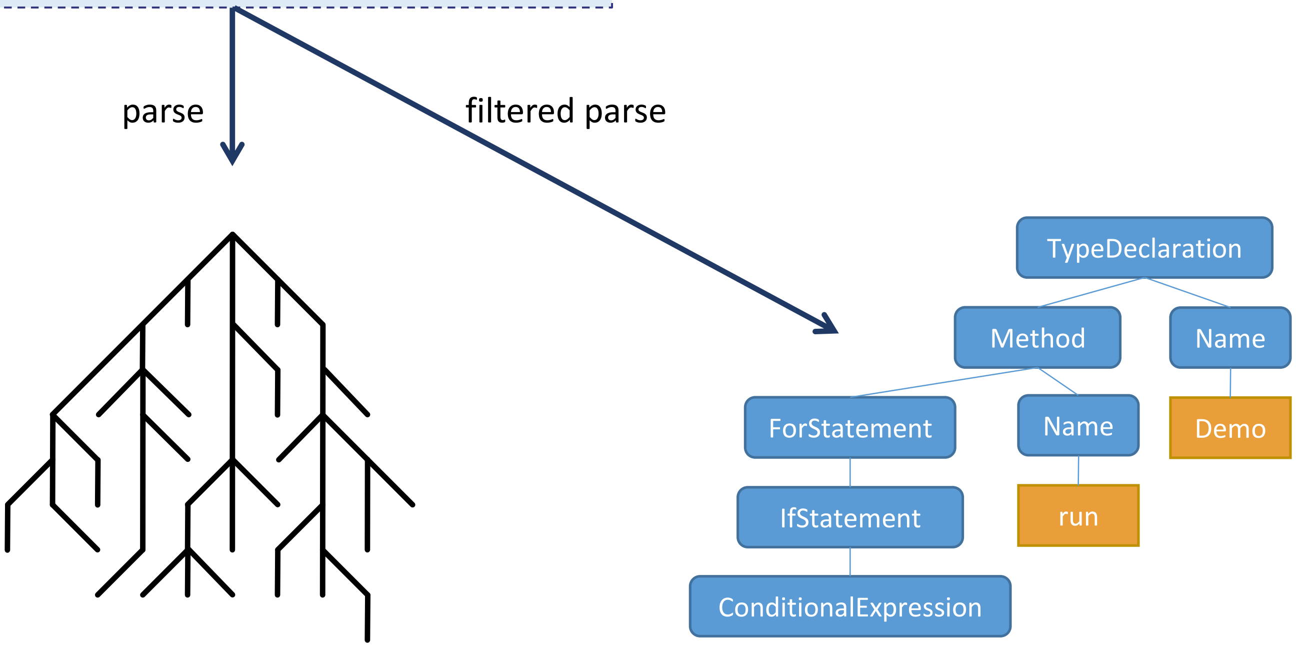
What's the complexity (1+#forks) and name for each method and class?



140 AST nodes (using ANTLR)

```
public class Demo {
  public void run() {
    for (int i = 1; i< 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
         System.out.println(i)
      }
  }
}
```

What's the complexity (1+#forks) and name for each method and class?



140 AST nodes
(using ANTLR)

7 AST nodes
(using ANTLR)

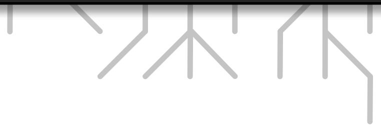
```
public class Demo {
  public void run() {
    for (int i = 1; i< 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
         System.out.println(i)
      }
    }
}
```

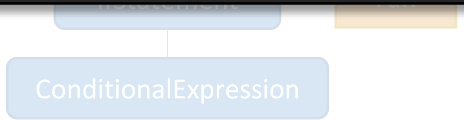
What's the complexity (1+#forks) and name for eachmethod and class?

parse

filtered parse

Storing only needed AST nodes applies a manyfold reduction in needed space





140 AST nodes
(using ANTLR)

7 AST nodes
(using ANTLR)

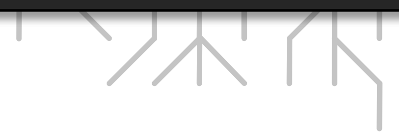
```
public class Demo {
  public void run() {
    for (int i = 1; i< 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
         System.out.println(i)
      }
    }
}
```

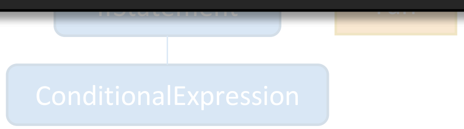
What's the complexity (1+#forks) and name for eachmethod and class?

parse

filtered parse

PS: Which AST nodes to load into the graph depends on the analysis

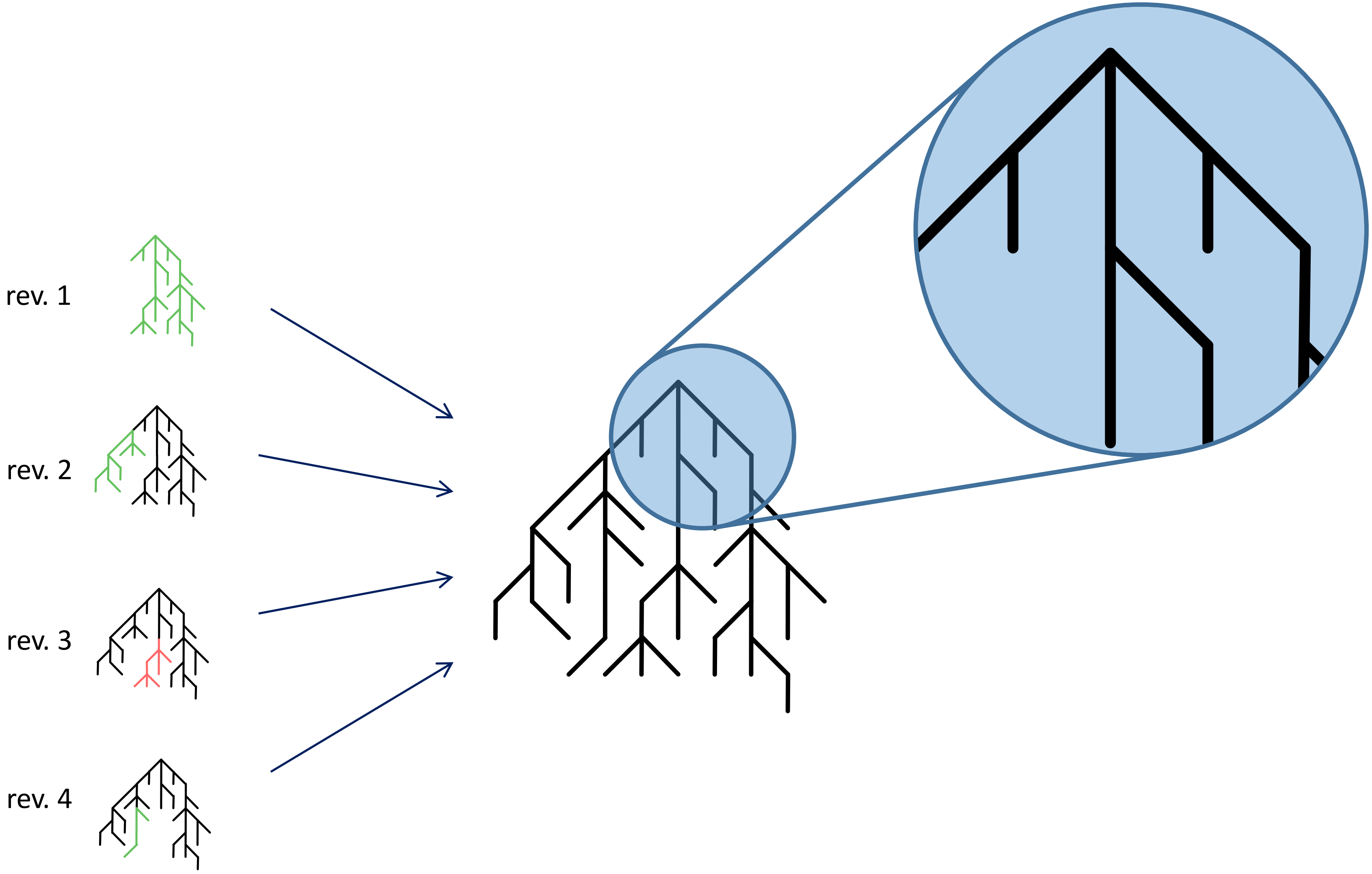


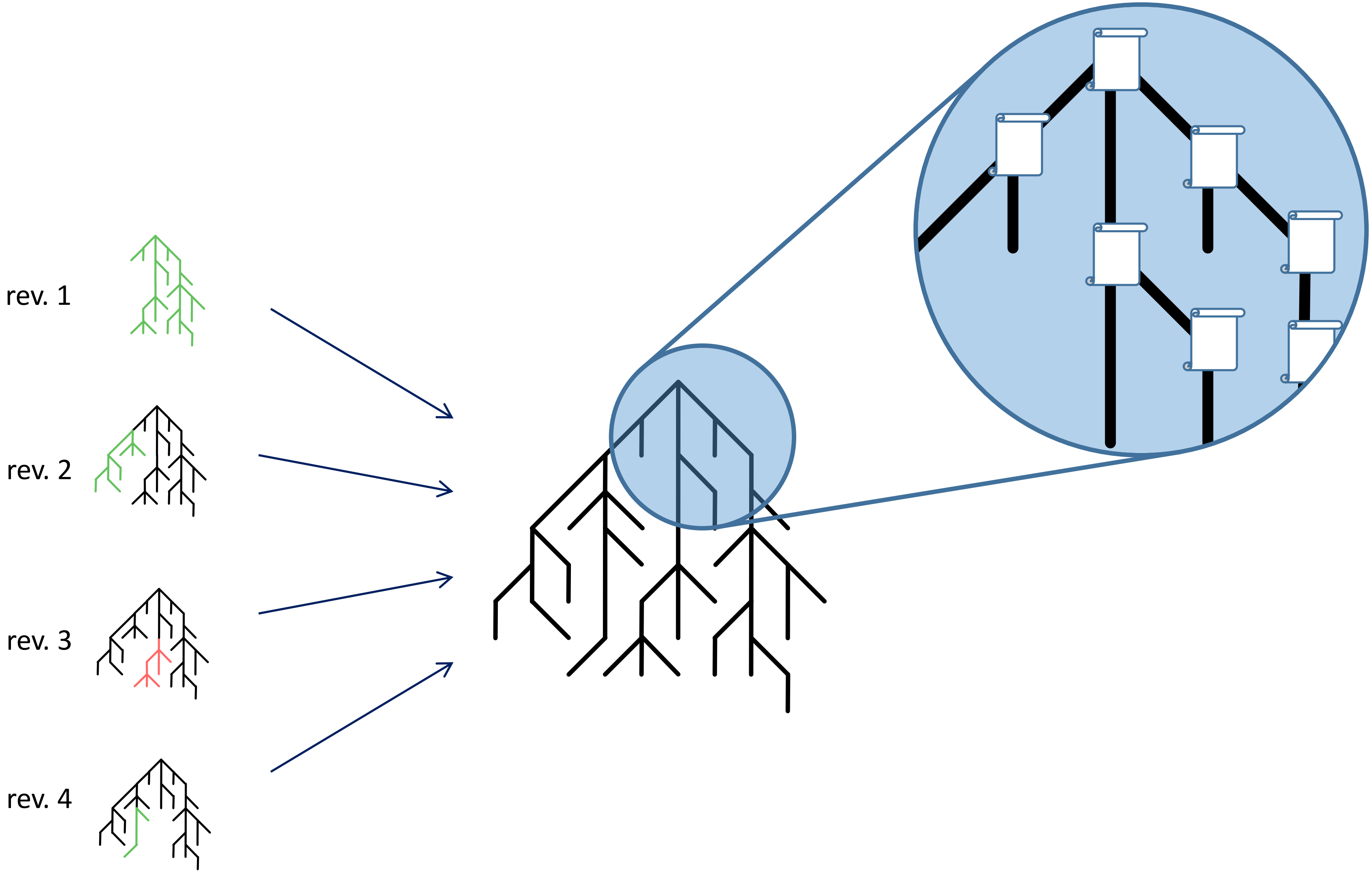


140 AST nodes
(using ANTLR)

7 AST nodes
(using ANTLR)

#4: Use non-duplicative data structures to store your results


rev. 1
rev. 2
rev. 3
rev. 4

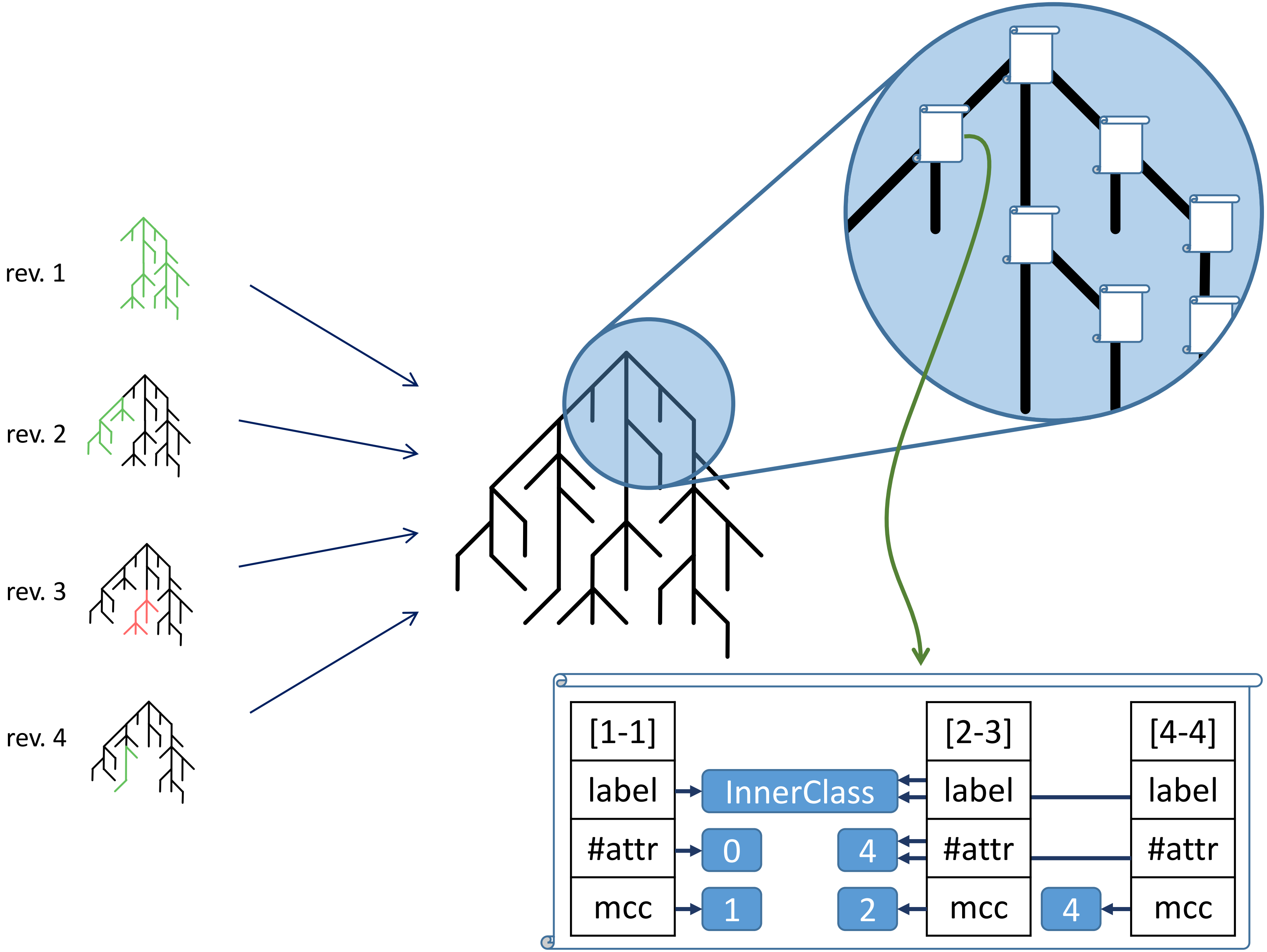


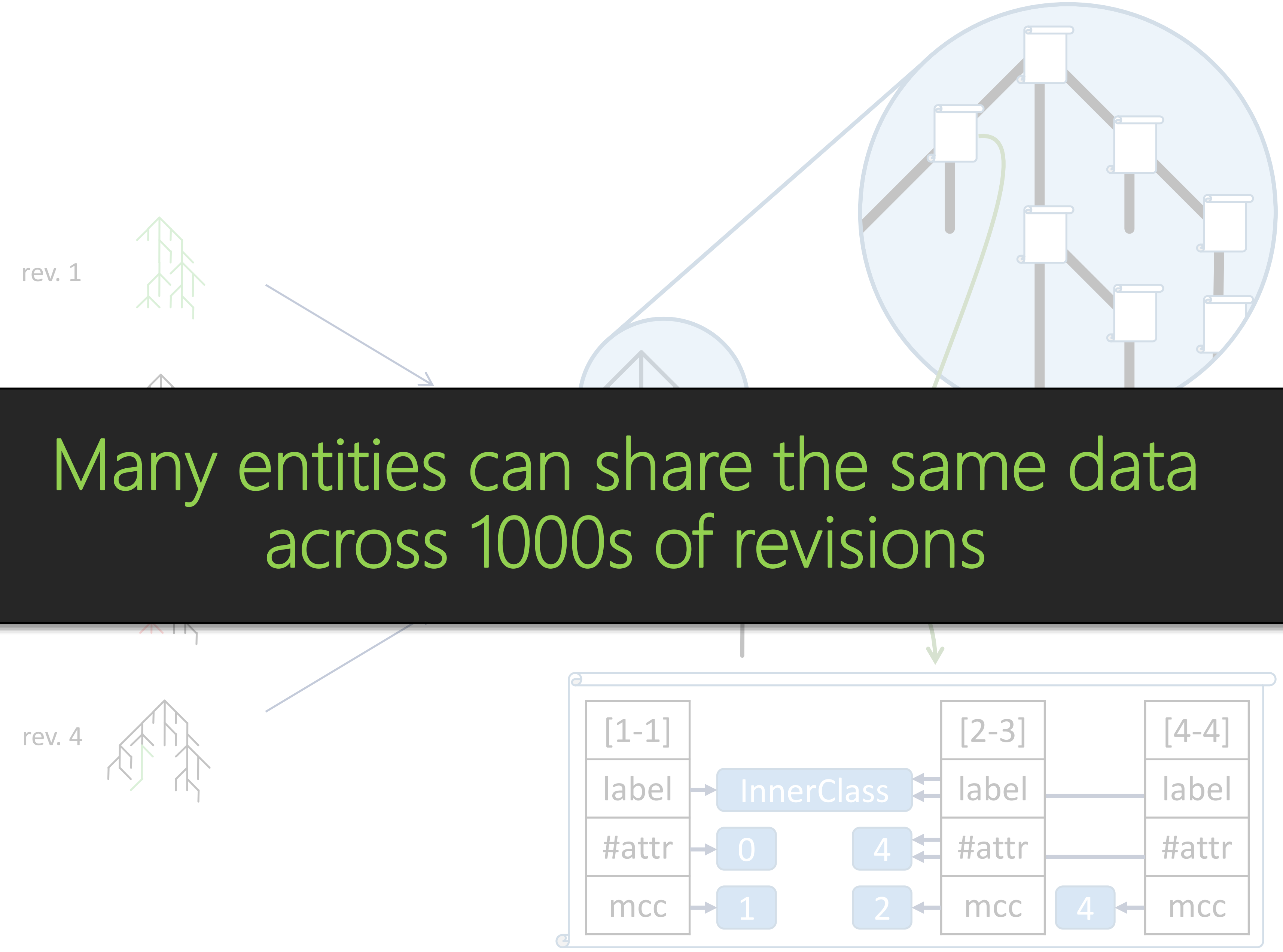
rev. 1

rev. 2

rev. 3

rev. 4


rev. 1
rev. 2
rev. 3
rev. 4
[1-1]
label
#attr
mcc
InnerClass
0
1
4
2
[2-3]
label
#attr
mcc
4
[4-4]
label
#attr
mcc



Many entities can share the same data across 1000s of revisions

LISA also does:

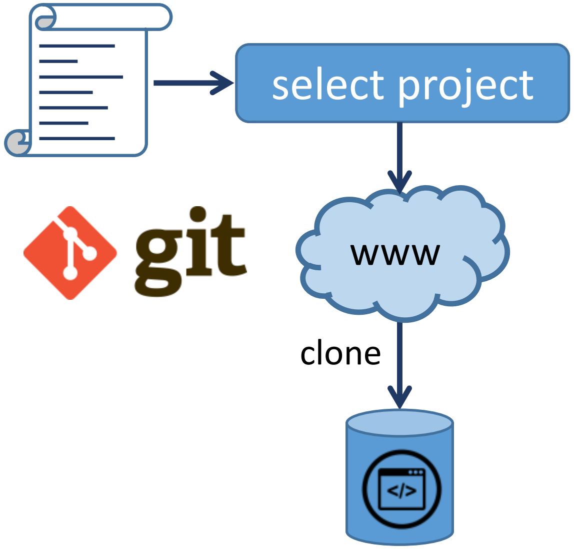
#5: Parallel Parsing

#6: Asynchronous graph computation

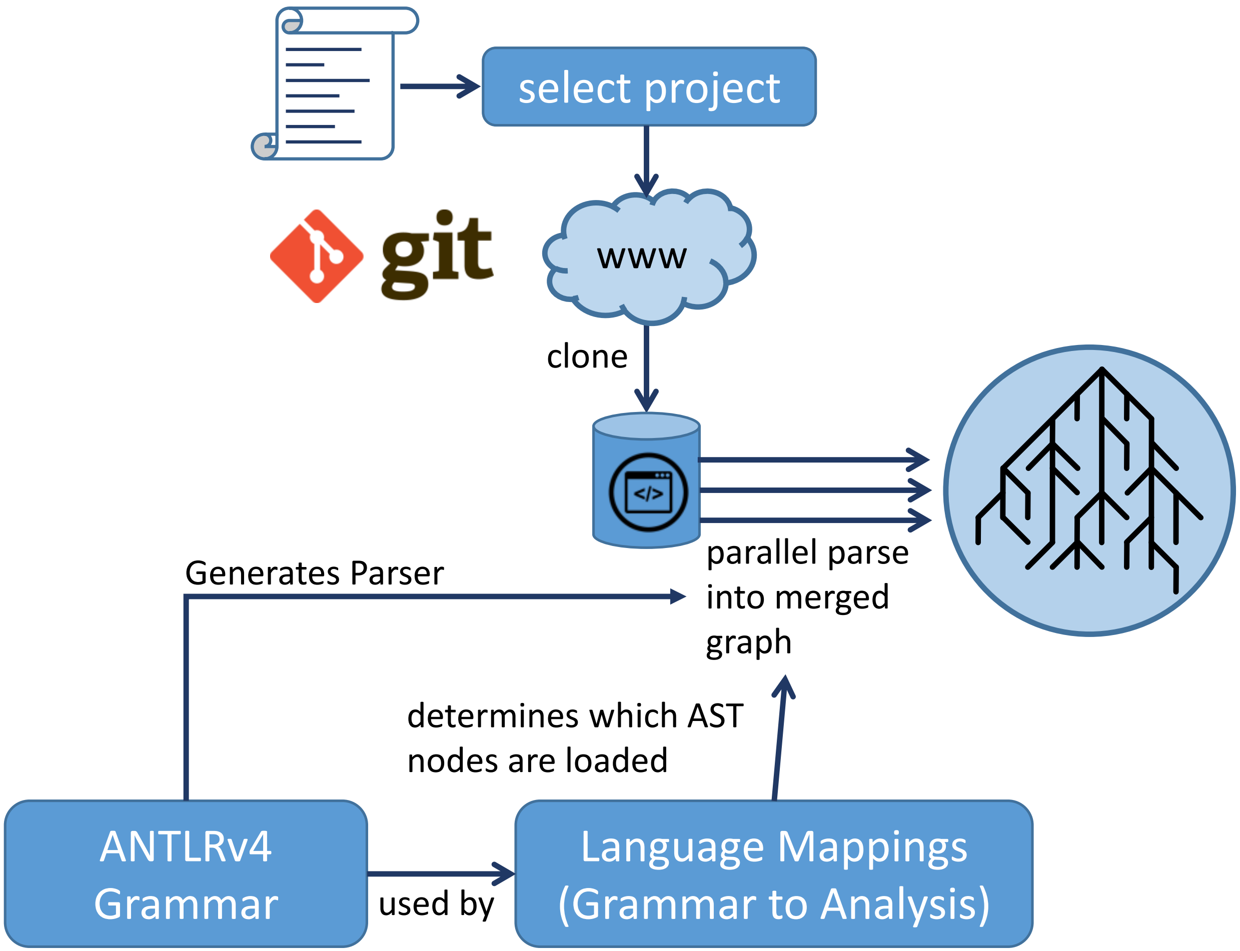
#7: **Generic graph computations** applying to ASTs from **compatible languages**

To Summarize…

The LISA Analysis Process



The LISA Analysis Process



select project

git

www

clone

parallel parse into merged graph

Generates Parser

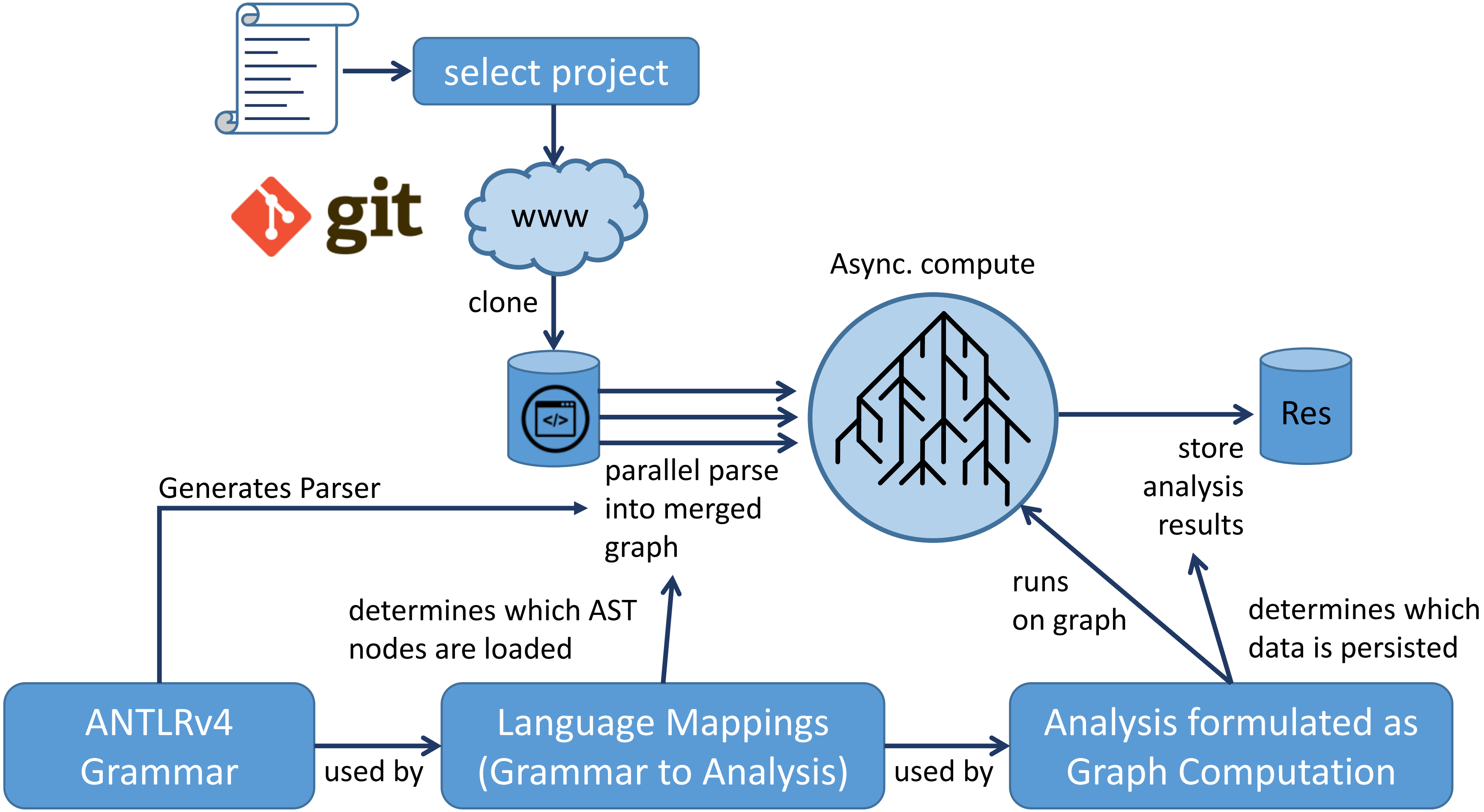
determines which AST nodes are loaded

ANTLRv4 Grammar

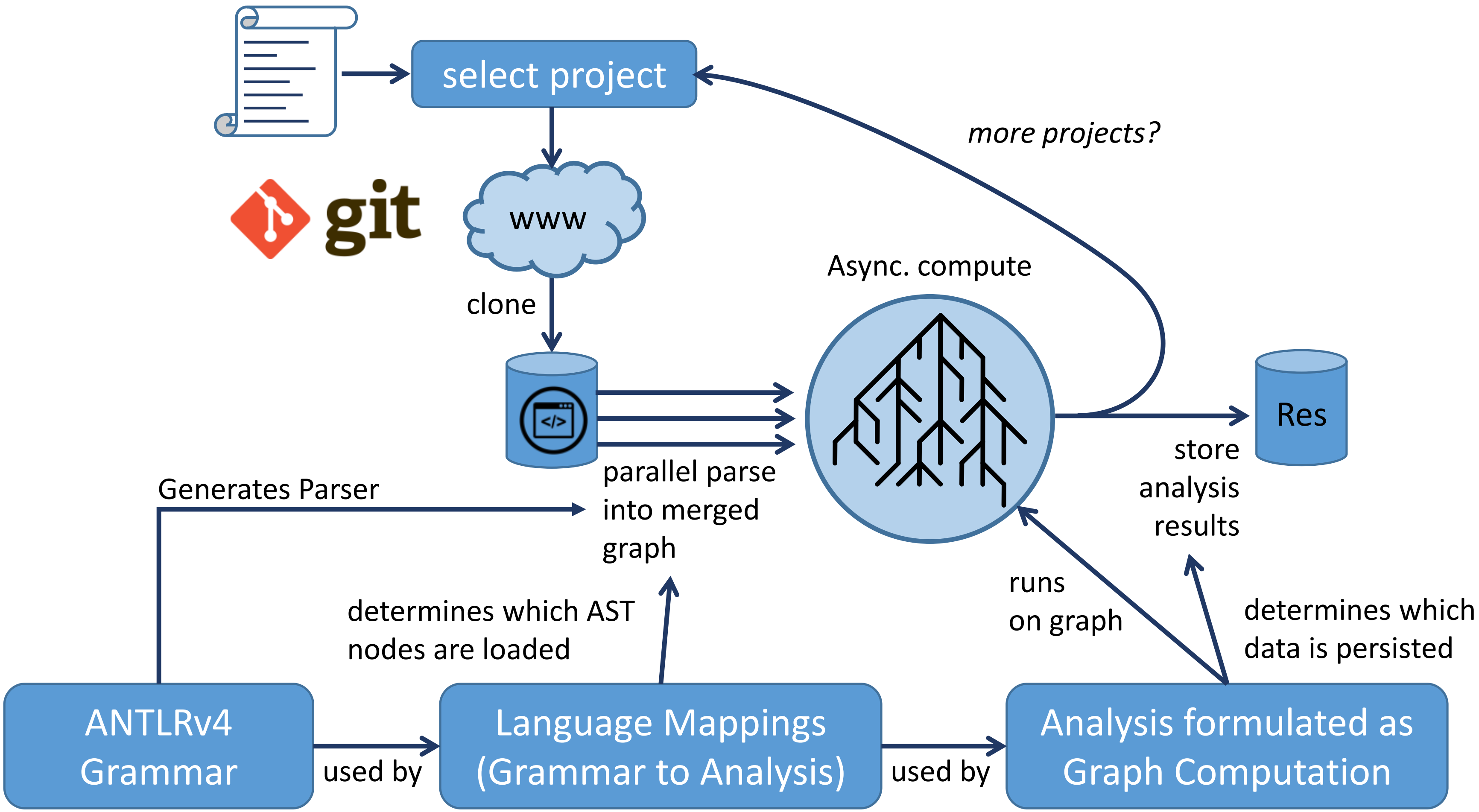
used by

Language Mappings (Grammar to Analysis)

The LISA Analysis Process



The LISA Analysis Process

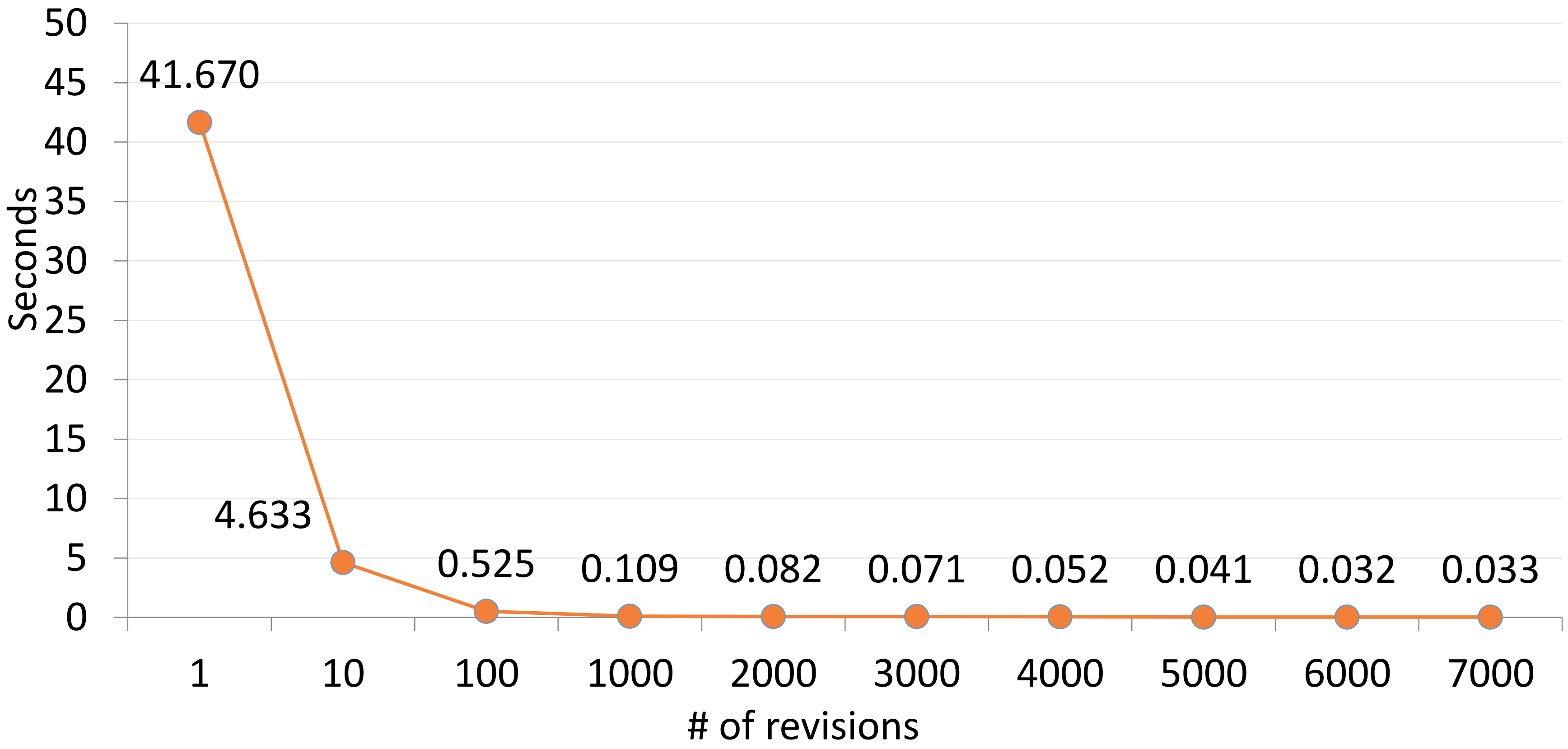

select project
git
www
clone
more projects?
Async. compute
Res
store analysis results
parallel parse into merged graph
Generates Parser
determines which AST nodes are loaded
runs on graph
determines which data is persisted
ANTLRv4 Grammar
used by
Language Mappings (Grammar to Analysis)
used by
Analysis formulated as Graph Computation

University of Zurich^UZH s.e.a.l. software evolution & architecture lab

How well does it work, then?

Marginal cost for +1 revision

Average Parsing+Computation time per Revision when analyzing n revisions of AspectJ (10 common metrics)



University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Overall Performance Stats

Language	Java	C#	JavScript
#Projects	100	100	100
#Revisions	646'261	489'764	204'301
#Files (parsed!)	3'235'852	3'234'178	507'612
#Lines (parsed!)	1'370'998'072	961'974'773	194'758'719
Total Runtime (RT)[1]	18:43h	52:12h	29:09h
Median RT[1]	2:15min	4:54min	3:43min
Tot. Avg. RT per Rev.[2]	84ms	401ms	531ms
Med. Avg. RT per Rev.[2]	**30ms**	**116ms**	**166ms**

[1] Including cloning and persisting results

[2] Excluding cloning and persisting results

What's the catch?

(There are a few...)

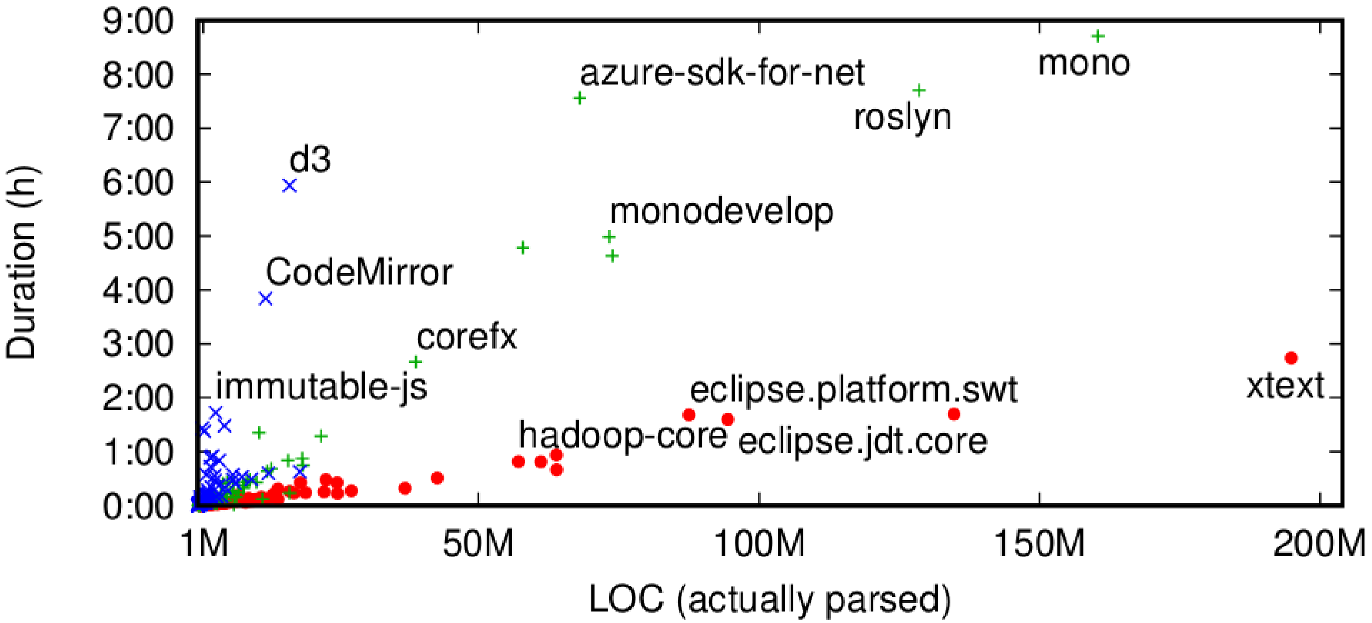
The (not so) minor stuff

- Must implement analyses from scratch
  - No help from a compiler
  - Non-file-local analyses need some effort

The (not so) minor stuff

- Must implement analyses from scratch
  - No help from a compiler
  - Non-file-local analyses need some effort
- Moved files/methods etc. add overhead
  - Uniquely identifying files/entities is hard
  - (No impact on results, though)

Language matters

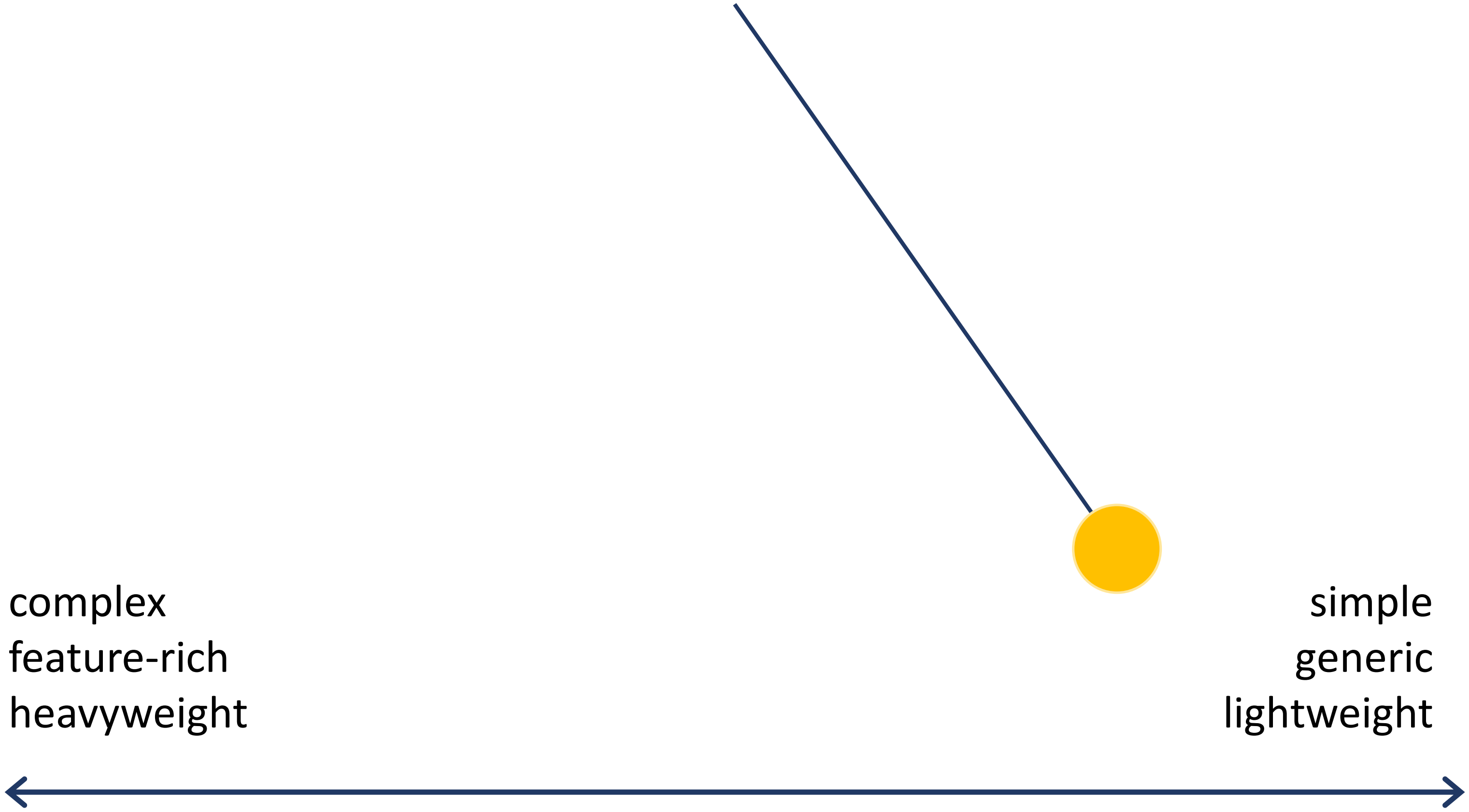


Javascript **C#** **Java**

E.g.: Javascript takes longer because:

- Larger files, less modularization
- Slower parser (automatic semicolon-insertion)

LISA is EXTREME



complex
feature-rich
heavyweight

simple
generic
lightweight

University of Zurich^UZH

s.e.a.l.
software evolution & architecture lab

Thank you for your attention

Read the paper: http://t.uzh.ch/Fj

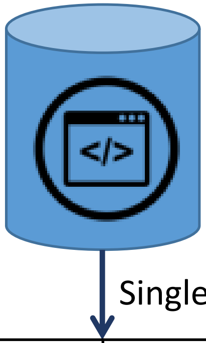
Try the tool: http://t.uzh.ch/Fk

Get the slides: http://t.uzh.ch/Fm

Contact me: alexandru@ifi.uzh.ch

SANER ‘17, Klagenfurt, 22.02.2017

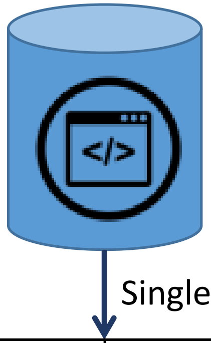
Parallelize Parsing



src/Main.java	{1: 1251a4}, {3: fc2452}, {4: 251929}
src/Settings.java	{2: fa255a}
src/Foo.java	{1: 512fc2}, {4: 791c2a}, {5: bcb215}
src/Bar.java	{4: 8a23b2}, {5: b2399f}

Obtain *sequence* of Git blob ids for old versions of each unique path

Parallelize Parsing



Single Git tree traversal

src/Main.java	{1: 1251a4}, {3: fc2452}, {4: 251929}
src/Settings.java	{2: fa255a}
src/Foo.java	{1: 512fc2}, {4: 791c2a}, {5: bcb215}
src/Bar.java	{4: 8a23b2}, {5: b2399f}

Obtain *sequence* of Git blob ids for old versions of each unique path

Parse files with different paths in parallel
Some files will have more revisions, taking longer to parse in total
→ Parsing only takes roughly as long as required for the file with the most revisions

Parallelize Parsing

Parallel Parsing from Git is easy and has no overhead

src/Foo.java	{1: 512fc2}, {4: 791c2a}, {5: bcb215}
src/Bar.java	{4: 8a23b2}, {5: b2399f}

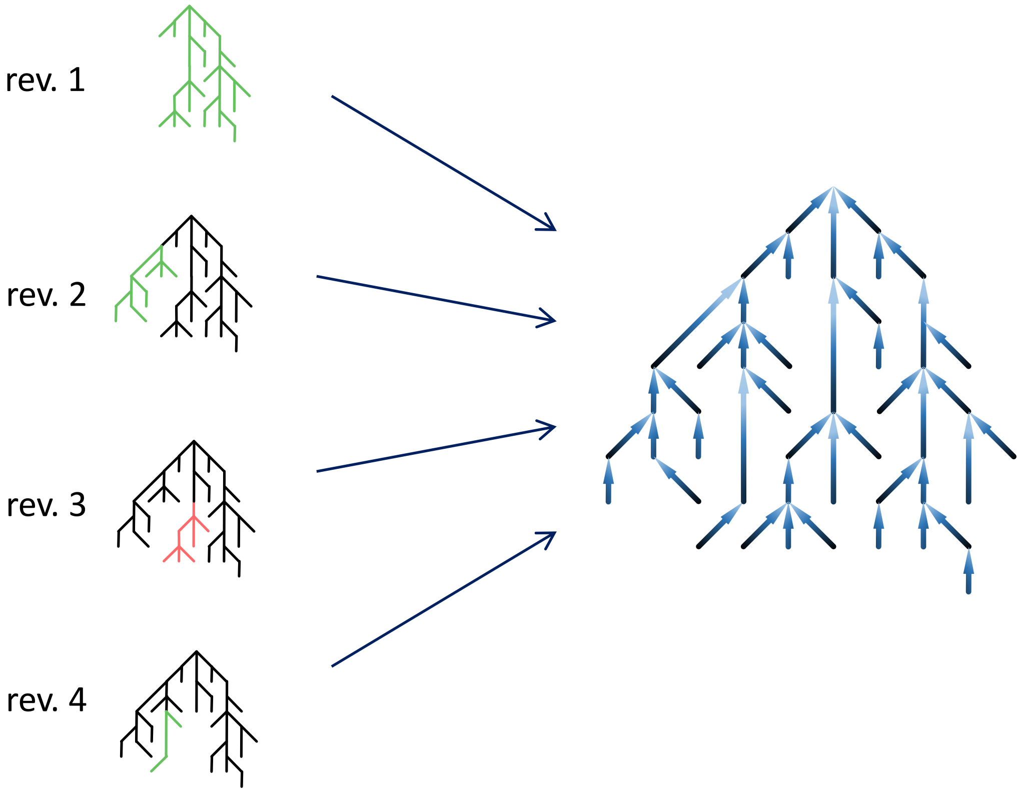
→ Parsing only takes roughly as long as required for the file with the most revisions

Obtain *sequence* of Git blob ids for old versions of each unique path

"Speed-up factor" for each technique

- Parallel parsing: Roughly 2
- Merged ASTs: >1000 for many revisions
- Filtered parsing: >10 during computation, depends on how much is filtered
- All depends on file sizes / parser speed

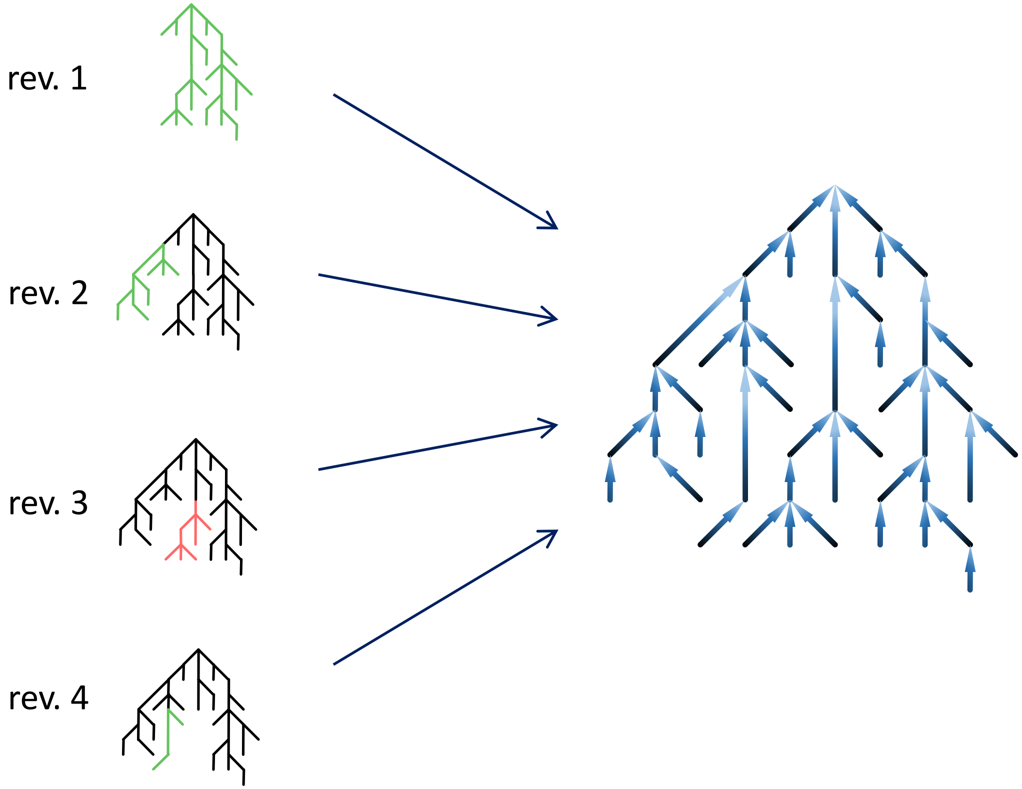
Asynchronous Graph Analysis



Depending on the node type:

- Signal specfic data

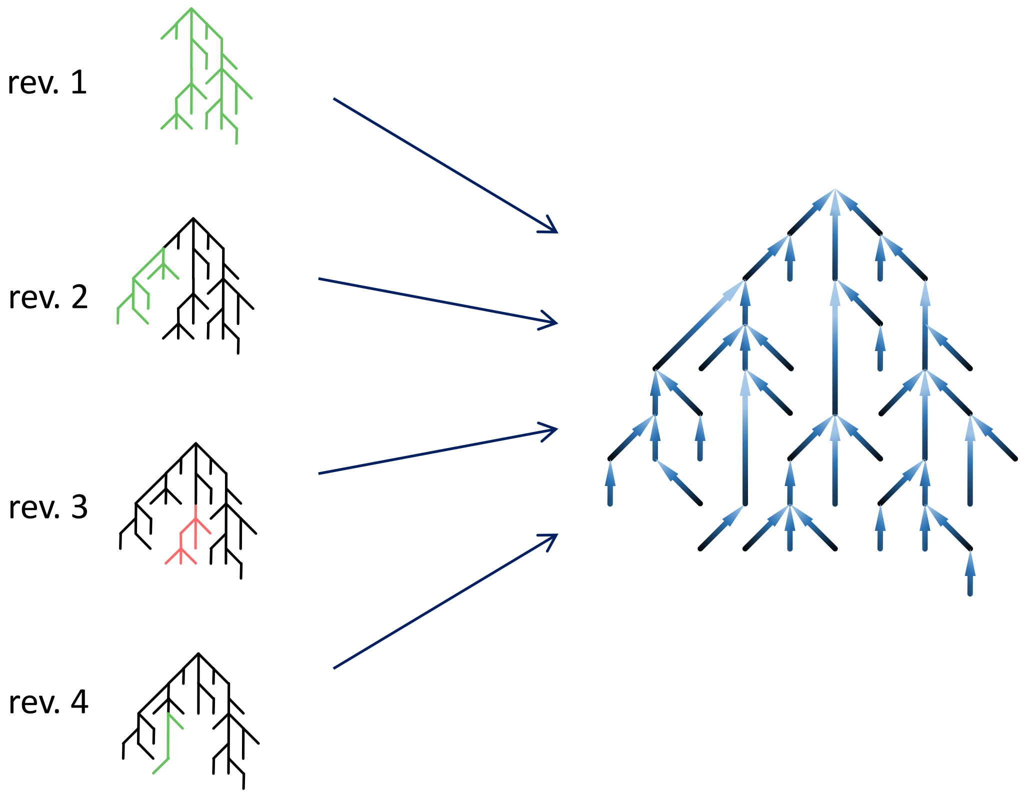
Asynchronous Graph Analysis



Depending on the node type:

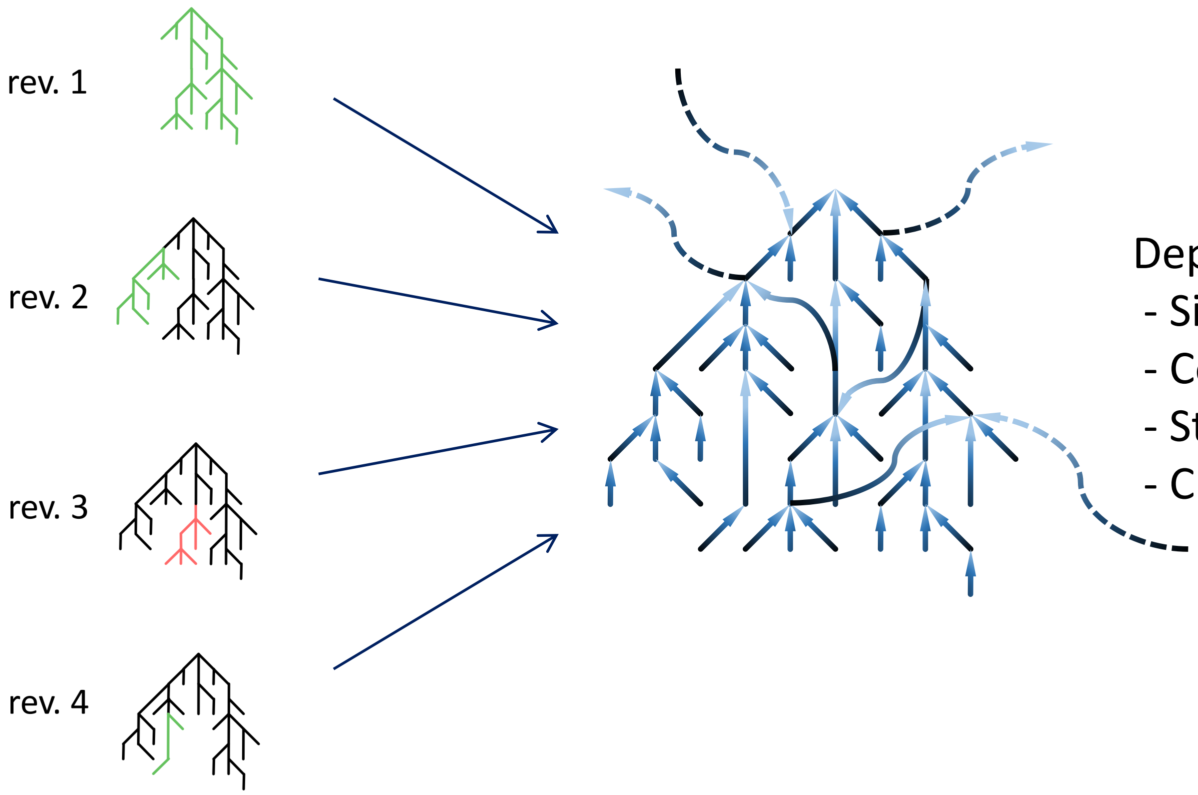
- Signal specfic data
- Collect specific data

Asynchronous Graph Analysis



Depending on the node type:
- Signal specfic data
- Collect specific data
- Store specific data

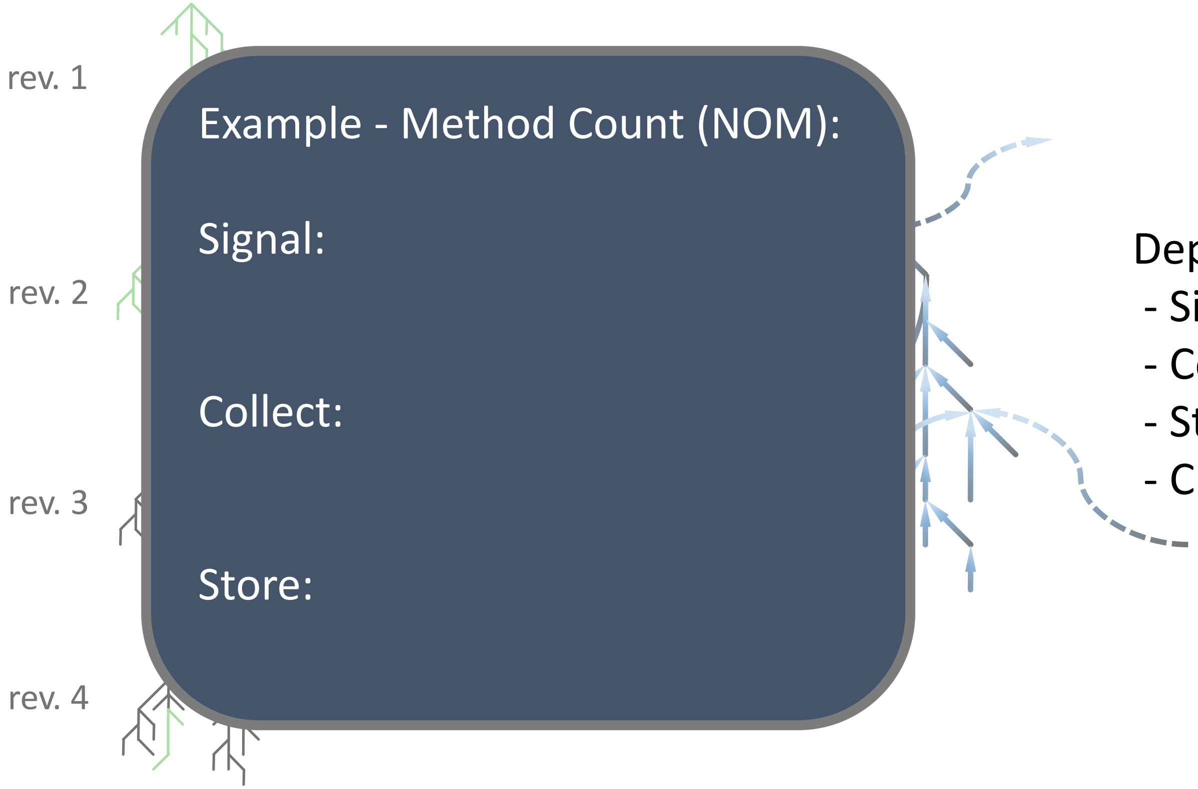
Asynchronous Graph Analysis



Depending on the node type:

- Signal specfic data
- Collect specific data
- Store specific data
- Create new nodes & edges

Asynchronous Graph Analysis



Example - Method Count (NOM):

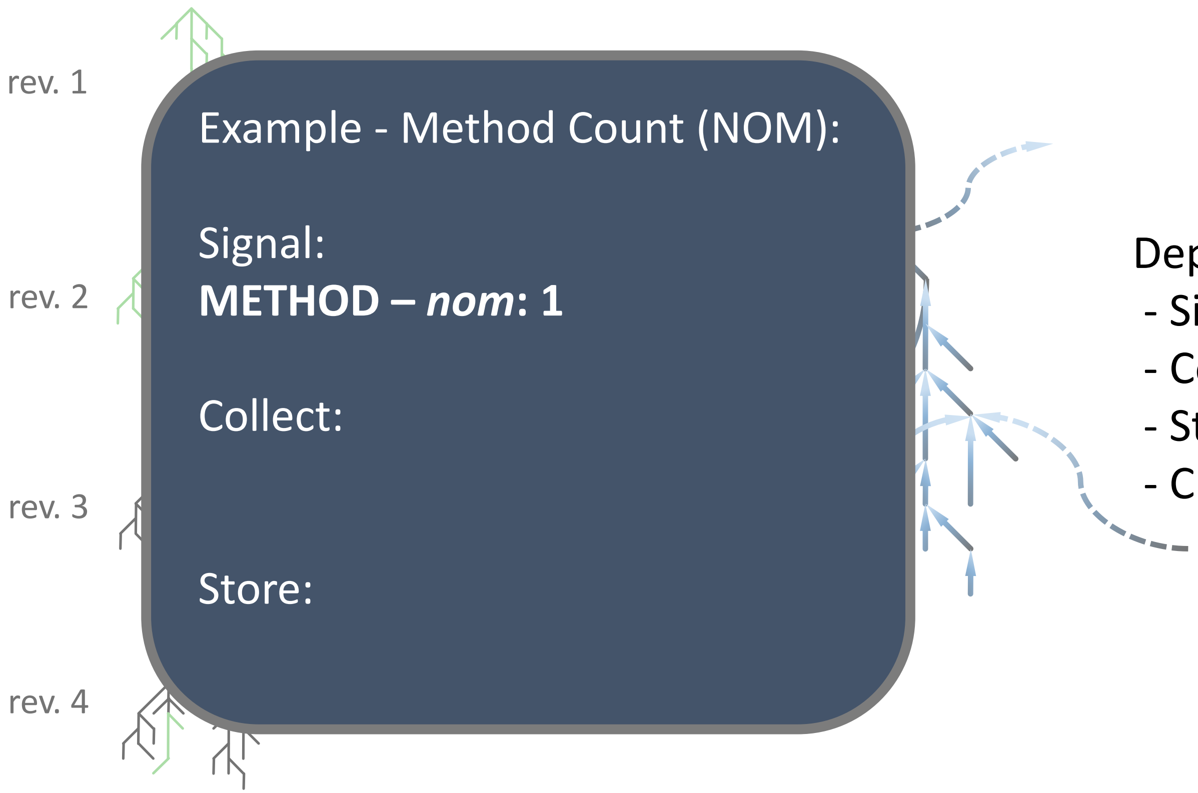
Signal:

Collect:

Store:

Depending on the node type:
- Signal specfic data
- Collect specific data
- Store specific data
- Create new nodes & edges

Asynchronous Graph Analysis



Example - Method Count (NOM):

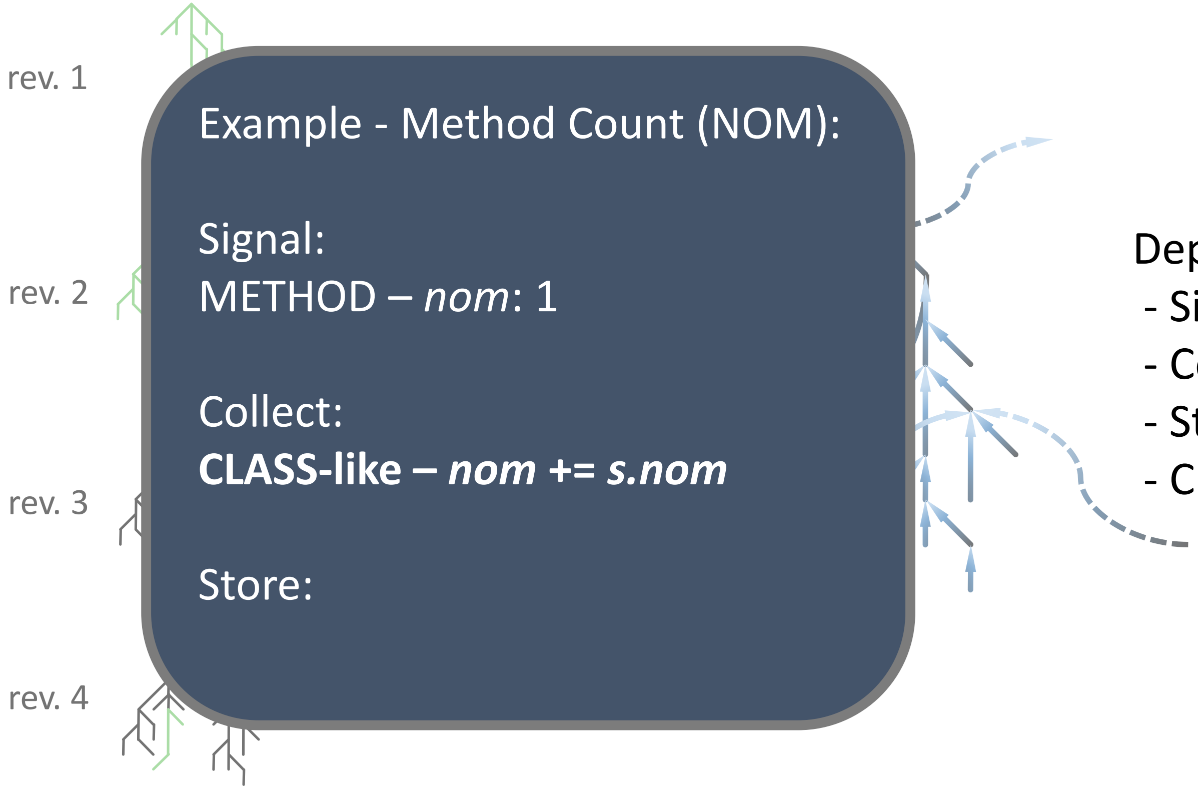
Signal:
METHOD – *nom*: 1

Collect:

Store:

Depending on the node type:
- Signal specfic data
- Collect specific data
- Store specific data
- Create new nodes & edges

Asynchronous Graph Analysis



Example - Method Count (NOM):

Signal:
METHOD – *nom*: 1

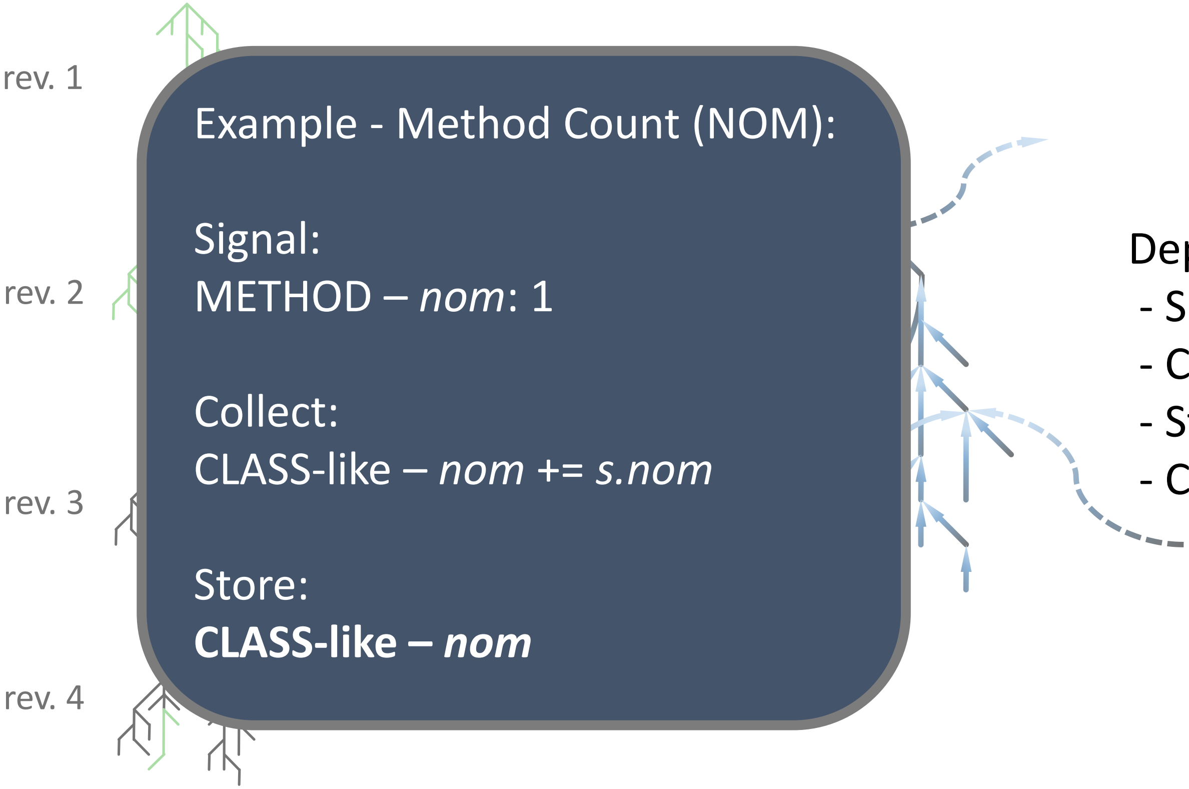
Collect:
CLASS-like – *nom += s.nom*

Store:

Depending on the node type:

- Signal specfic data
- Collect specific data
- Store specific data
- Create new nodes & edges

Asynchronous Graph Analysis



Example - Method Count (NOM):

Signal:
METHOD – *nom*: 1

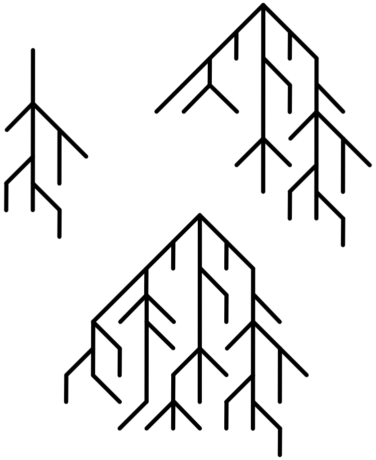
Collect:
CLASS-like – *nom += s.nom*

Store:
CLASS-like – *nom*

Depending on the node type:
- Signal specfic data
- Collect specific data
- Store specific data
- Create new nodes & edges

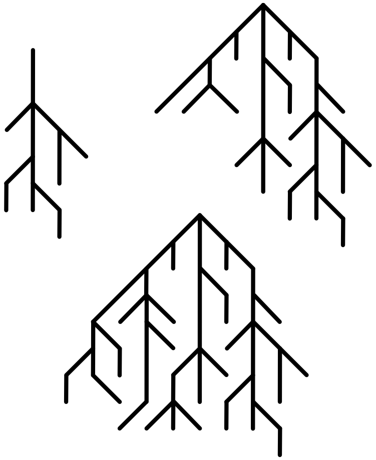
Static source code analysis?

Static source code analysis?



Static source code analysis?

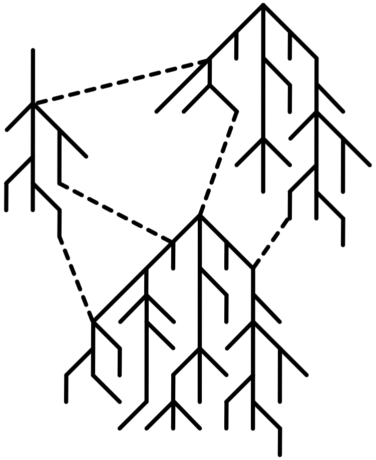
Simple Code Metrics
(NOC, NOM, WMC, Complexity, ...)



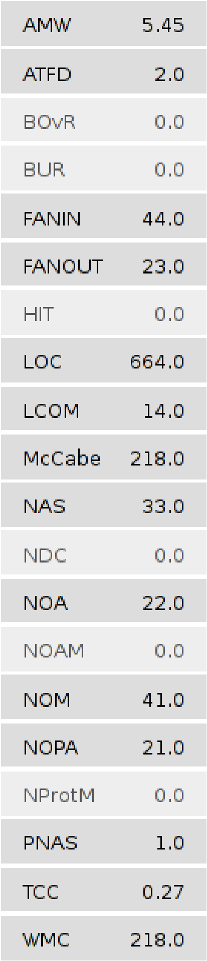
Static source code analysis?

Simple Code Metrics
(NOC, NOM, WMC, Complexity, ...)

Structure
(Coupling, Inheritance, ...)



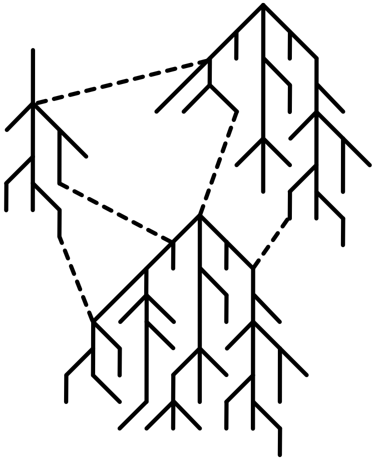
Static source code analysis?



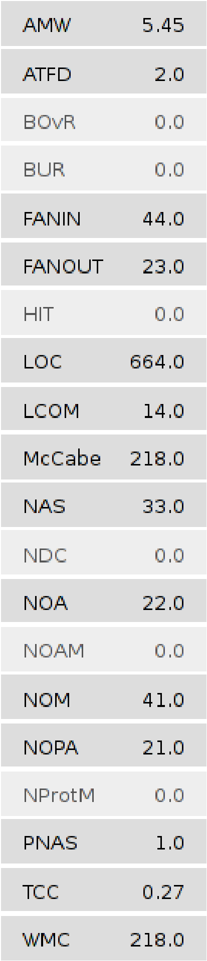
AMW	5.45
ATFD	2.0
BOvR	0.0
BUR	0.0
FANIN	44.0
FANOUT	23.0
HIT	0.0
LOC	664.0
LCOM	14.0
McCabe	218.0
NAS	33.0
NDC	0.0
NOA	22.0
NOAM	0.0
NOM	41.0
NOPA	21.0
NProtM	0.0
PNAS	1.0
TCC	0.27
WMC	218.0

Simple Code Metrics
(NOC, NOM, WMC, Complexity, ...)

Structure
(Coupling, Inheritance, ...)



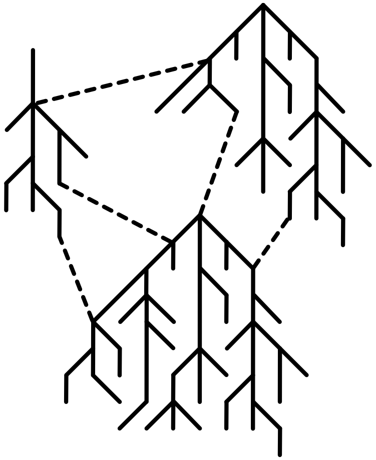
Static source code analysis?



AMW	5.45
ATFD	2.0
BOvR	0.0
BUR	0.0
FANIN	44.0
FANOUT	23.0
HIT	0.0
LOC	664.0
LCOM	14.0
McCabe	218.0
NAS	33.0
NDC	0.0
NOA	22.0
NOAM	0.0
NOM	41.0
NOPA	21.0
NProtM	0.0
PNAS	1.0
TCC	0.27
WMC	218.0

Simple Code Metrics
(NOC, NOM, WMC, Complexity, ...)

Structure
(Coupling, Inheritance, ...)

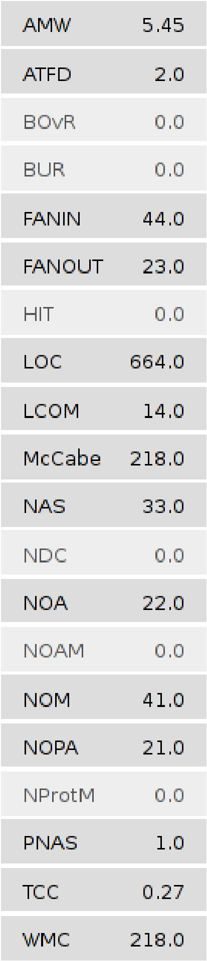


Practice
code smells
refactoring advice
hot-spot detection
bug prediction

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

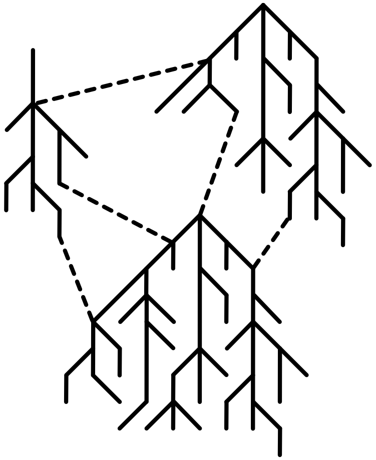
Static source code analysis?



AMW	5.45
ATFD	2.0
BOvR	0.0
BUR	0.0
FANIN	44.0
FANOUT	23.0
HIT	0.0
LOC	664.0
LCOM	14.0
McCabe	218.0
NAS	33.0
NDC	0.0
NOA	22.0
NOAM	0.0
NOM	41.0
NOPA	21.0
NProtM	0.0
PNAS	1.0
TCC	0.27
WMC	218.0

Simple Code Metrics
(NOC, NOM, WMC, Complexity, ...)

Structure
(Coupling, Inheritance, ...)



Practice
code smells
refactoring advice
hot-spot detection
bug prediction

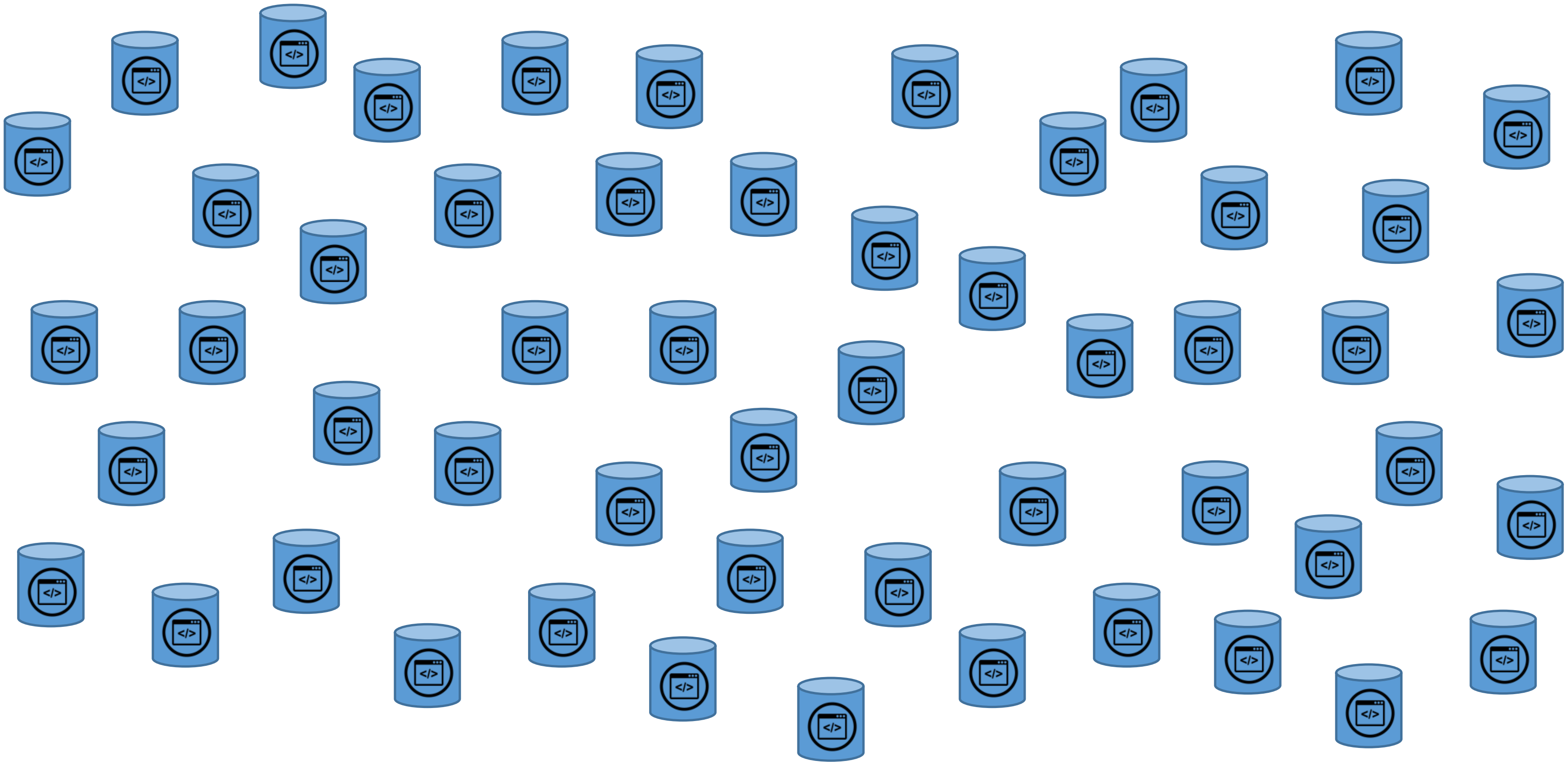
Research
understanding software evolution
identifying patterns &
anti-patterns
code quality assessment techniques
...
→ code studies

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

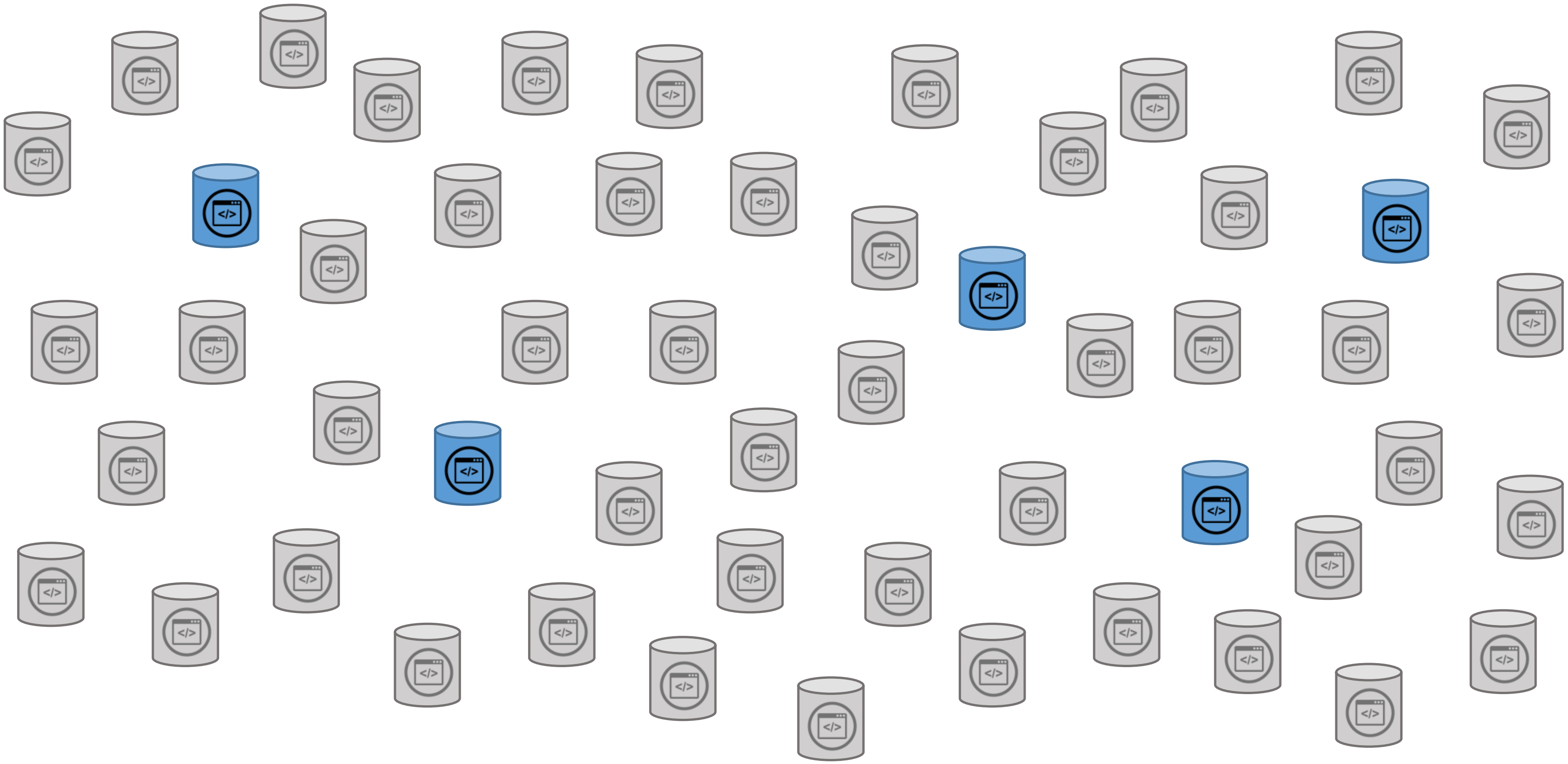
(Many) existing studies

(Many) existing studies



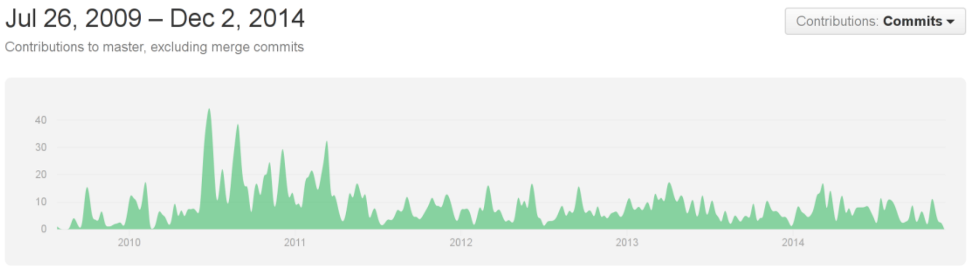
(Many) existing studies

- investigate a small number of projects

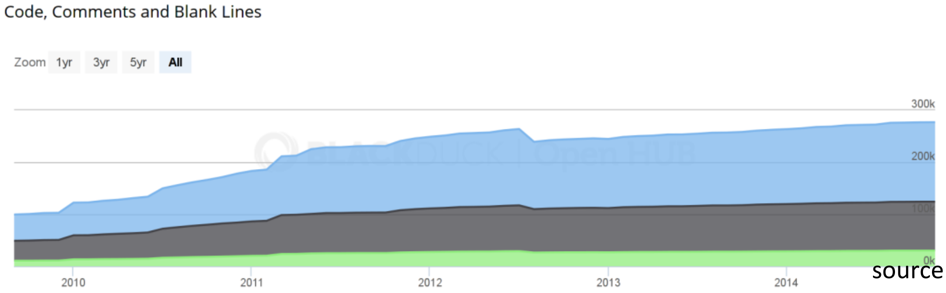


(Many) existing studies

- investigate a small number of projects



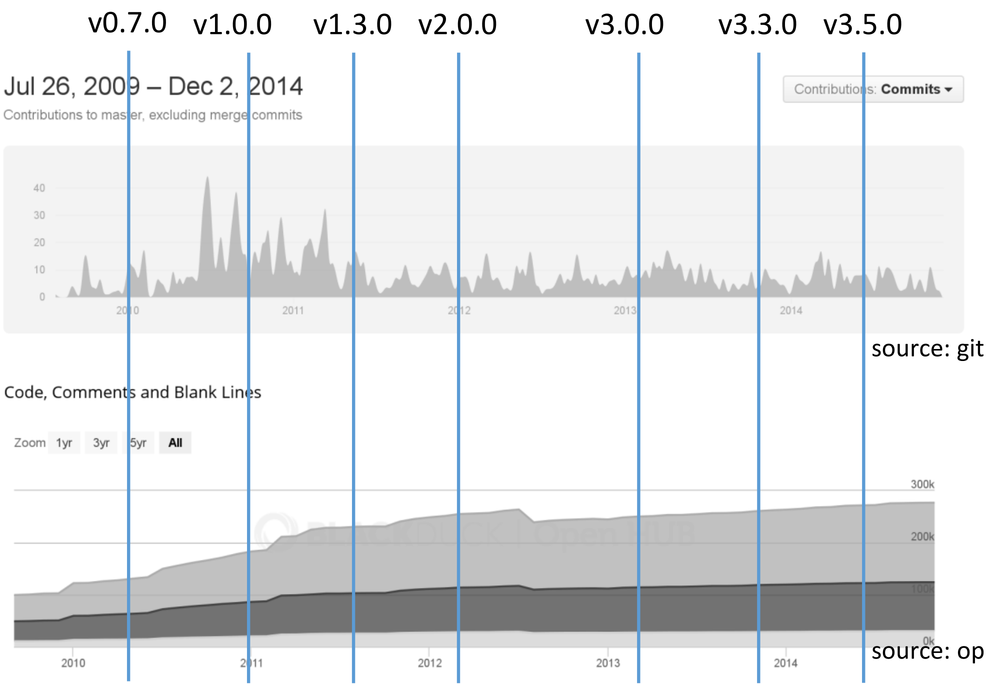
source: github.com



source: openhub.net

(Many) existing studies

- investigate a small number of projects
- analyze a few snapshots of multi-year projects

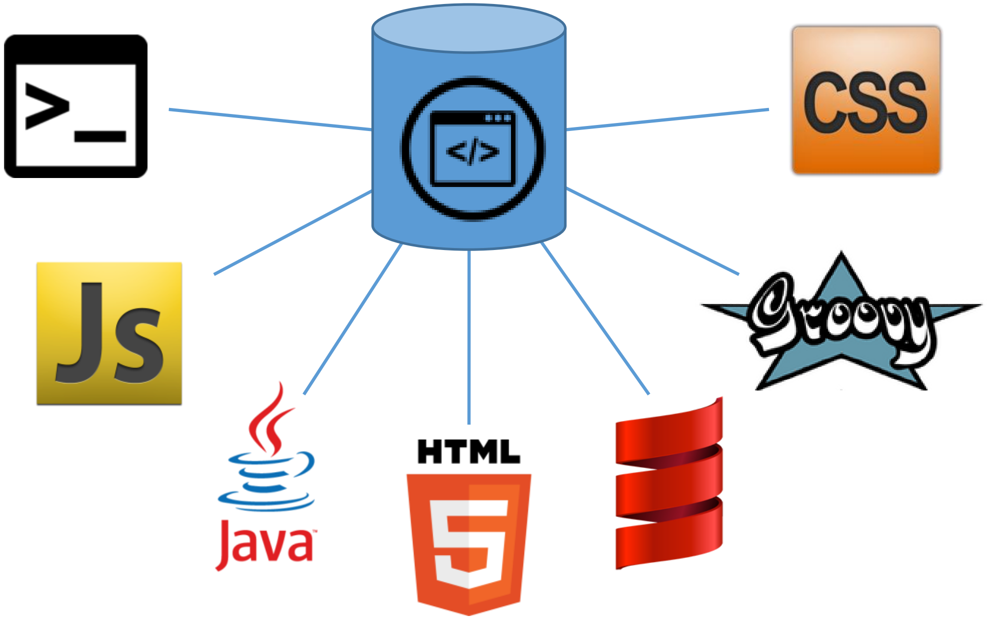


source: github.com

source: openhub.net

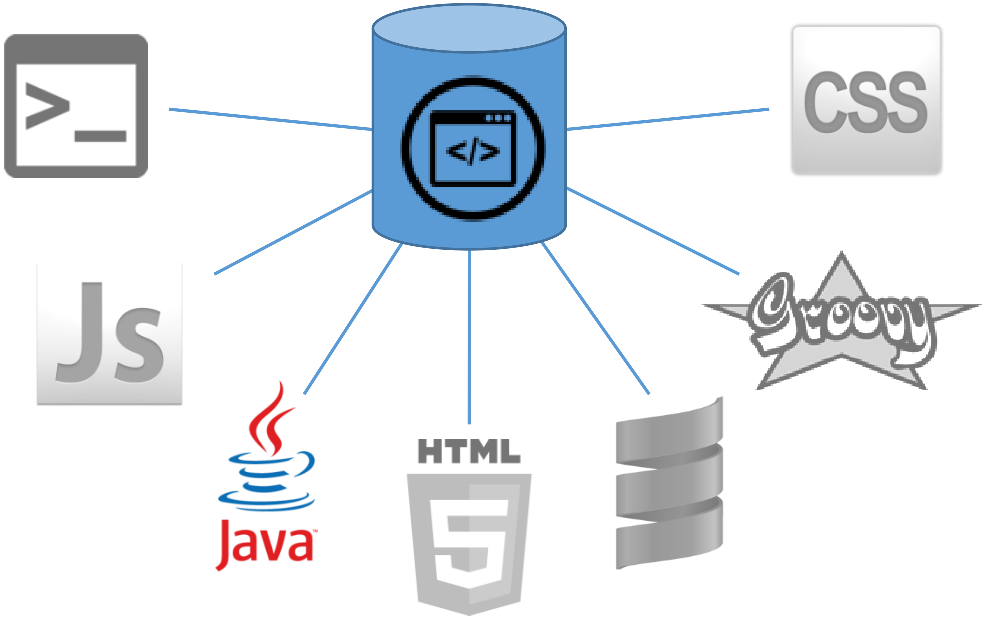
(Many) existing studies

- investigate a small number of projects
- analyze a few snapshots of multi-year projects



(Many) existing studies

- investigate a small number of projects
- analyze a few snapshots of multi-year projects
- focus on very few programming languages



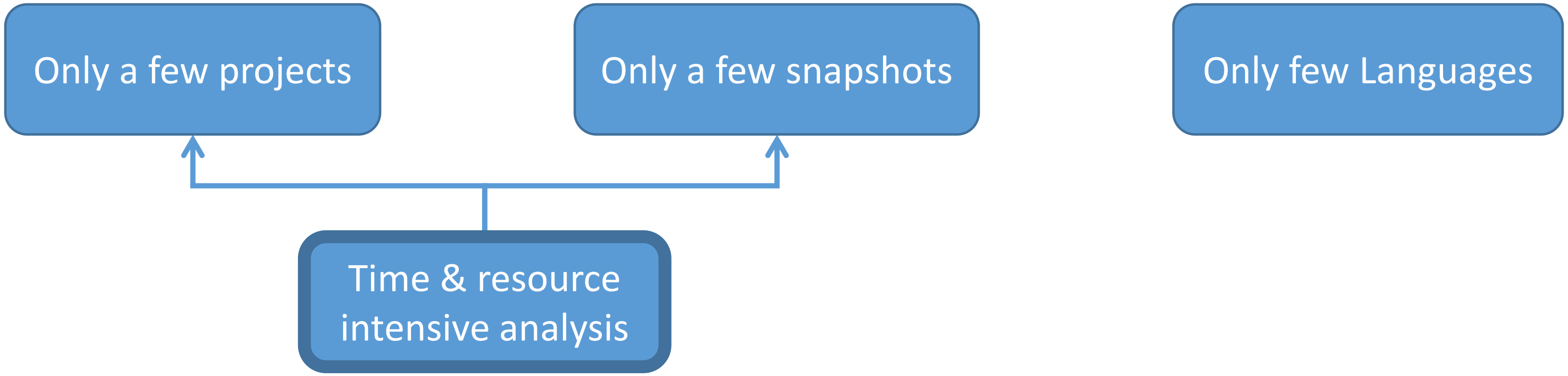
Why?

Only a few projects

Only a few snapshots

Only few Languages

Why?



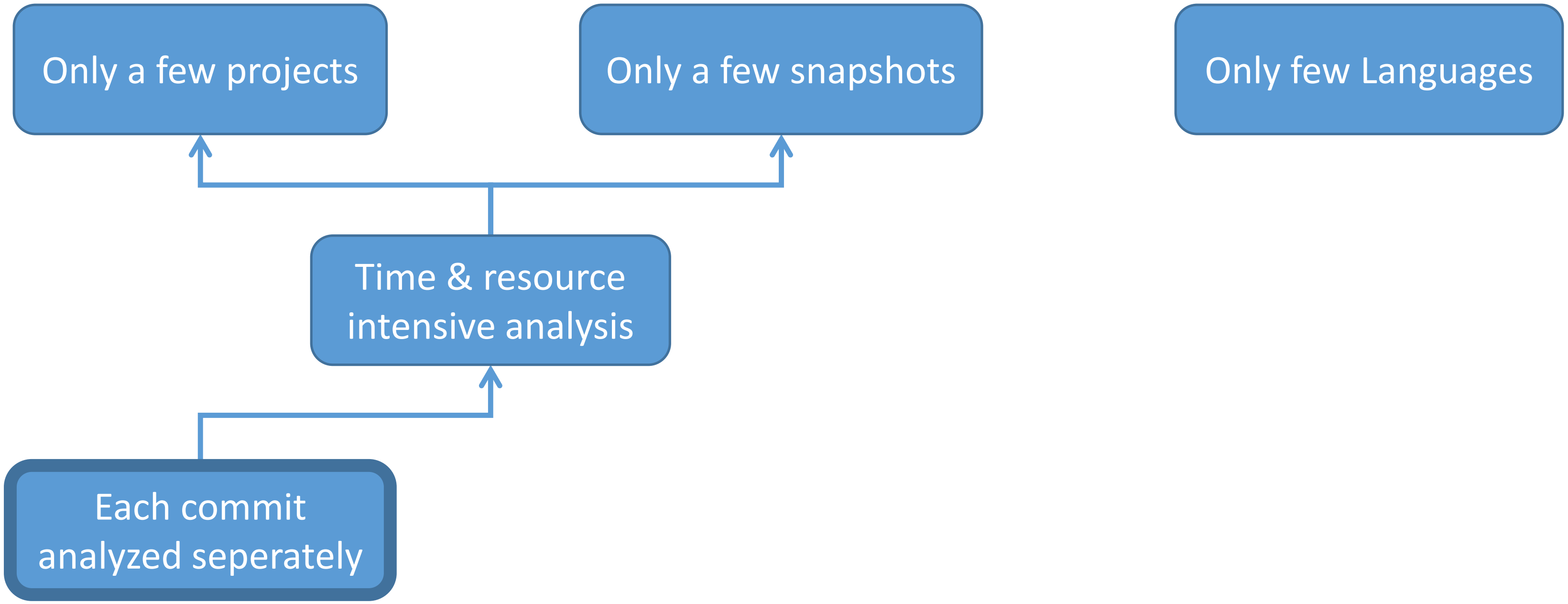
Only a few projects

Only a few snapshots

Only few Languages

Time & resource intensive analysis

Why?



Only a few projects

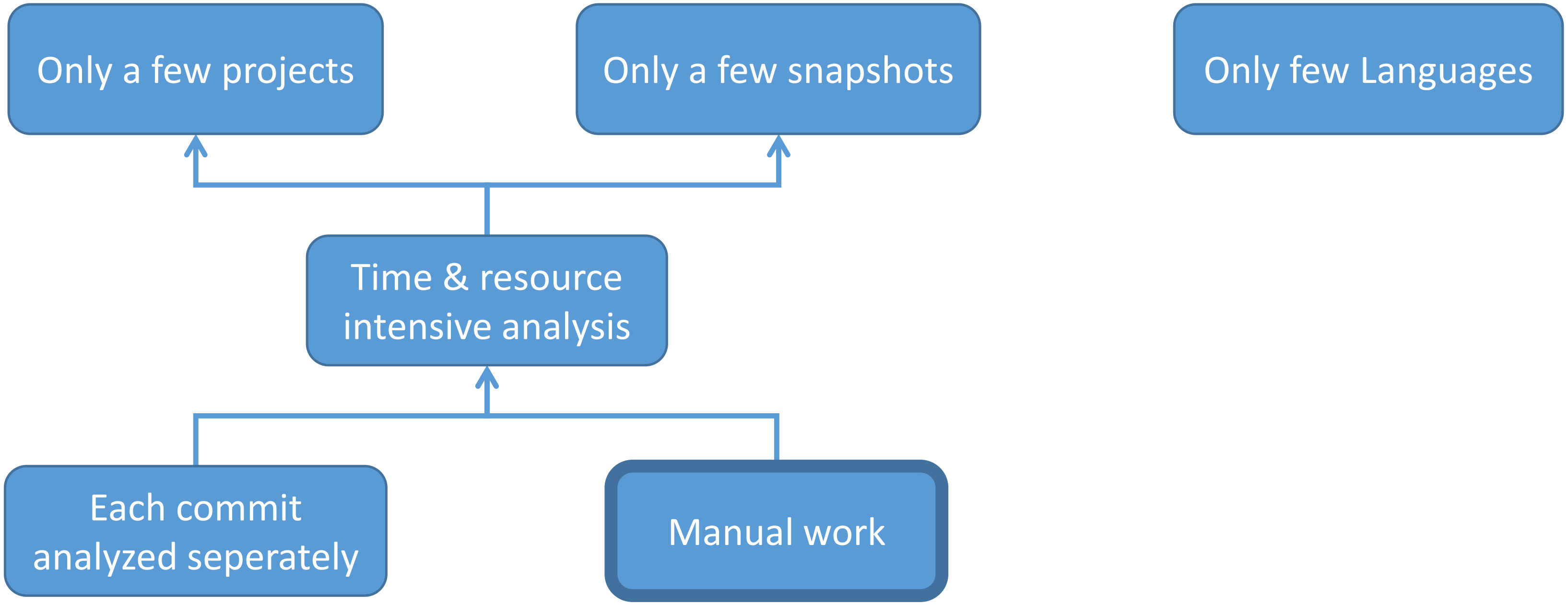
Only a few snapshots

Only few Languages

Time & resource intensive analysis

Each commit analyzed seperately

Why?


Only a few projects

Only a few snapshots

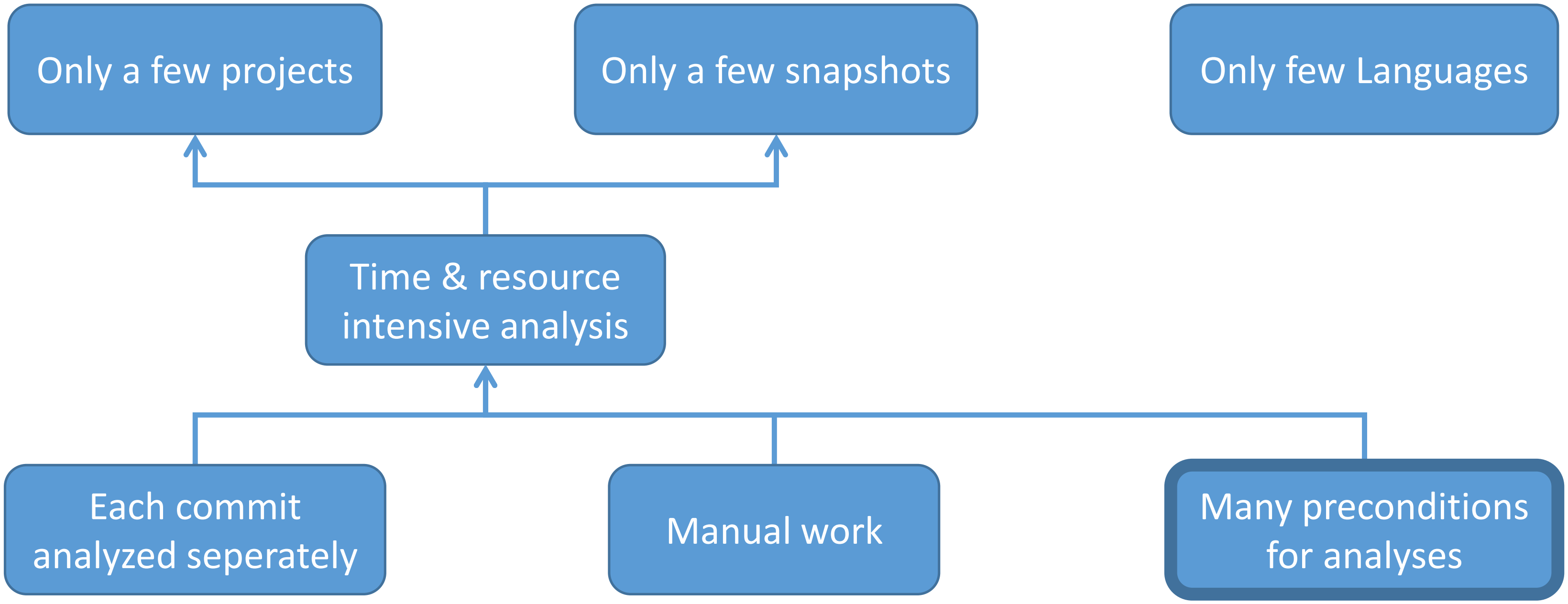
Only few Languages

Time & resource intensive analysis

Each commit analyzed seperately

Manual work

Why?



Only a few projects

Only a few snapshots

Only few Languages

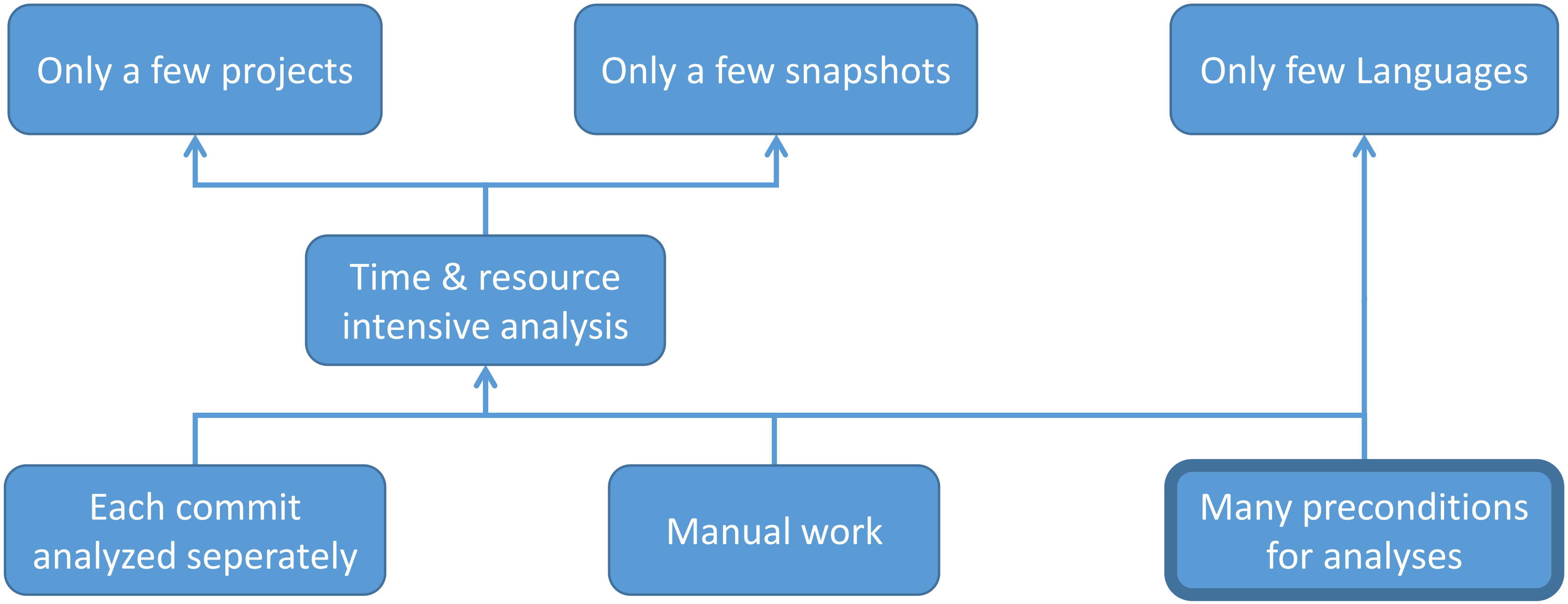
Time & resource intensive analysis

Each commit analyzed seperately

Manual work

Many preconditions for analyses

Why?


Only a few projects

Only a few snapshots

Only few Languages

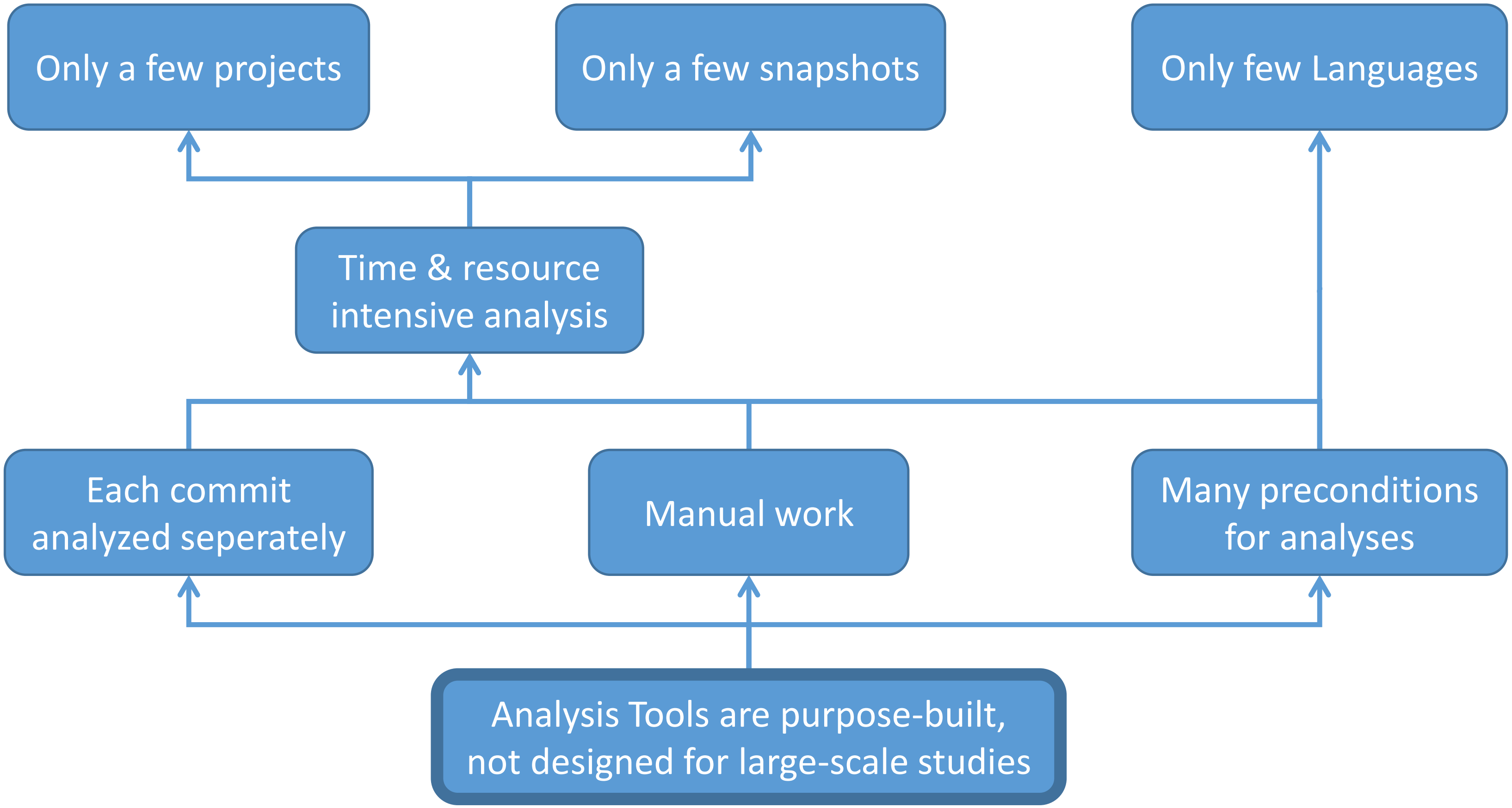
Time & resource intensive analysis

Each commit analyzed seperately

Manual work

Many preconditions for analyses

Why?



Only a few projects

Only a few snapshots

Only few Languages

Time & resource intensive analysis

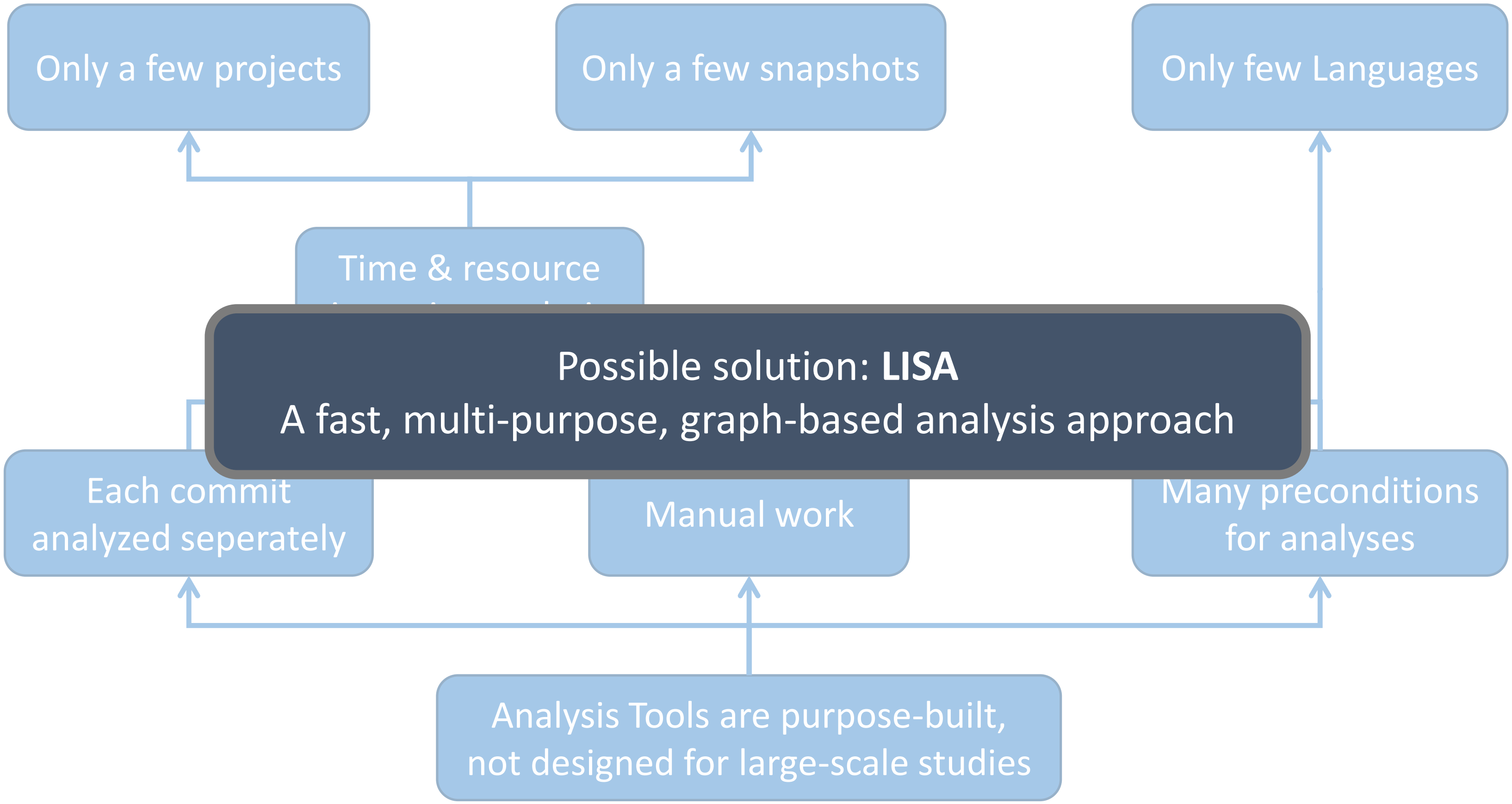
Each commit analyzed seperately

Manual work

Many preconditions for analyses

Analysis Tools are purpose-built, not designed for large-scale studies

Why?



Only a few projects

Only a few snapshots

Only few Languages

Time & resource

Each commit analyzed seperately

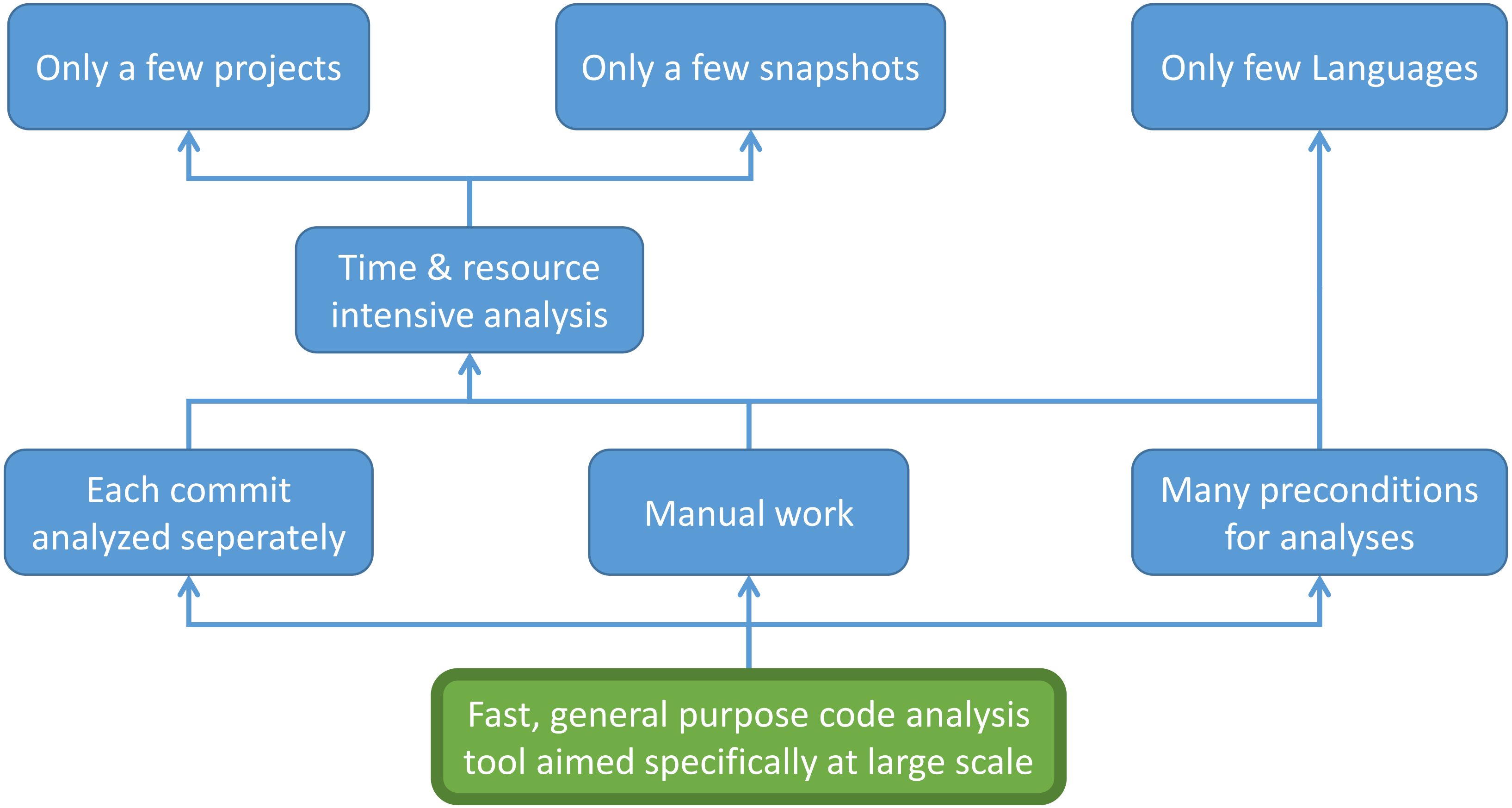
Manual work

Many preconditions for analyses

Analysis Tools are purpose-built, not designed for large-scale studies

Possible solution: **LISA**
A fast, multi-purpose, graph-based analysis approach

Rapid Analysis using LISA


Only a few projects

Only a few snapshots

Only few Languages

Time & resource intensive analysis

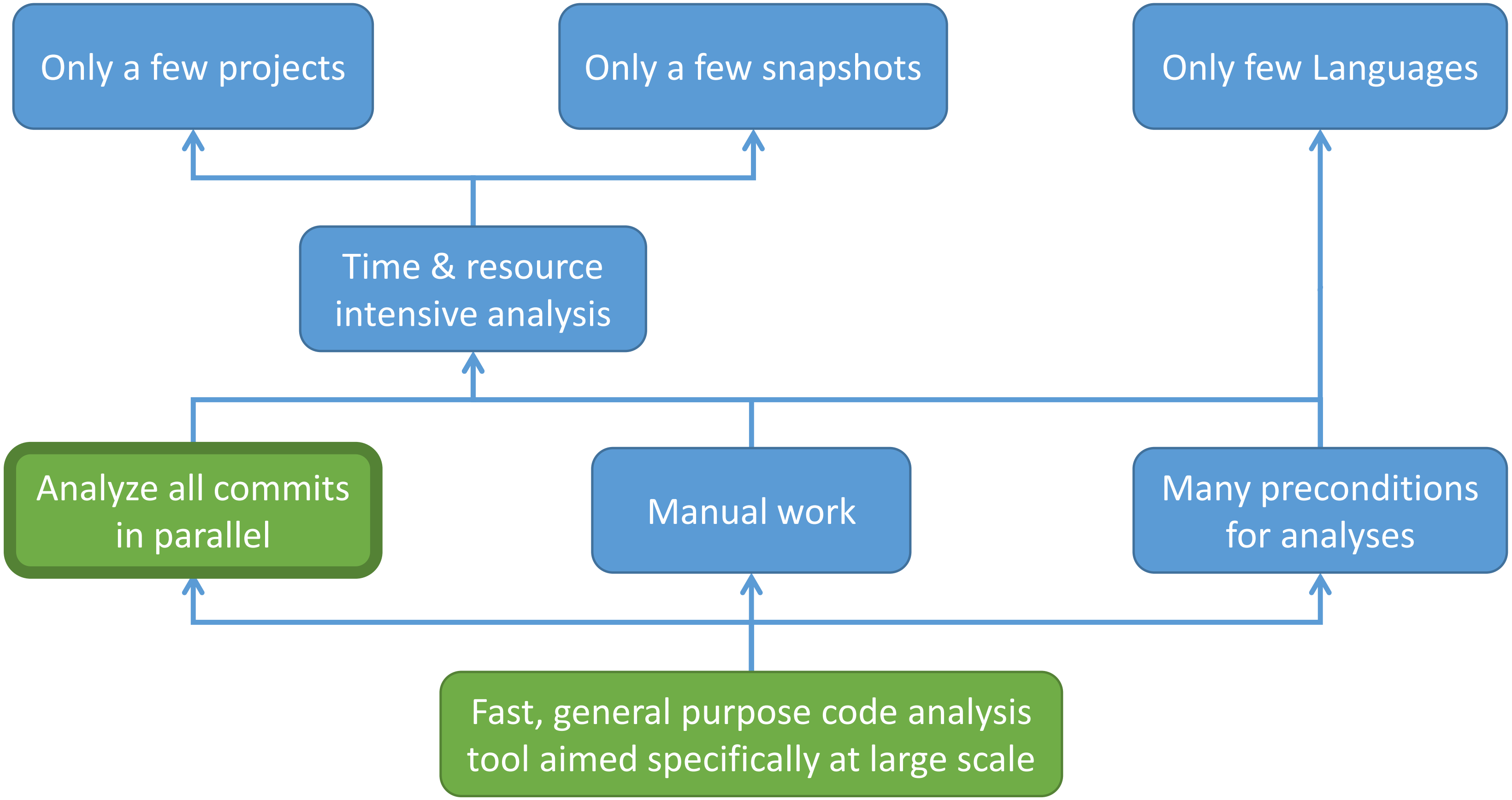
Each commit analyzed seperately

Manual work

Many preconditions for analyses

Fast, general purpose code analysis tool aimed specifically at large scale

Rapid Analysis using LISA



Only a few projects

Only a few snapshots

Only few Languages

Time & resource intensive analysis

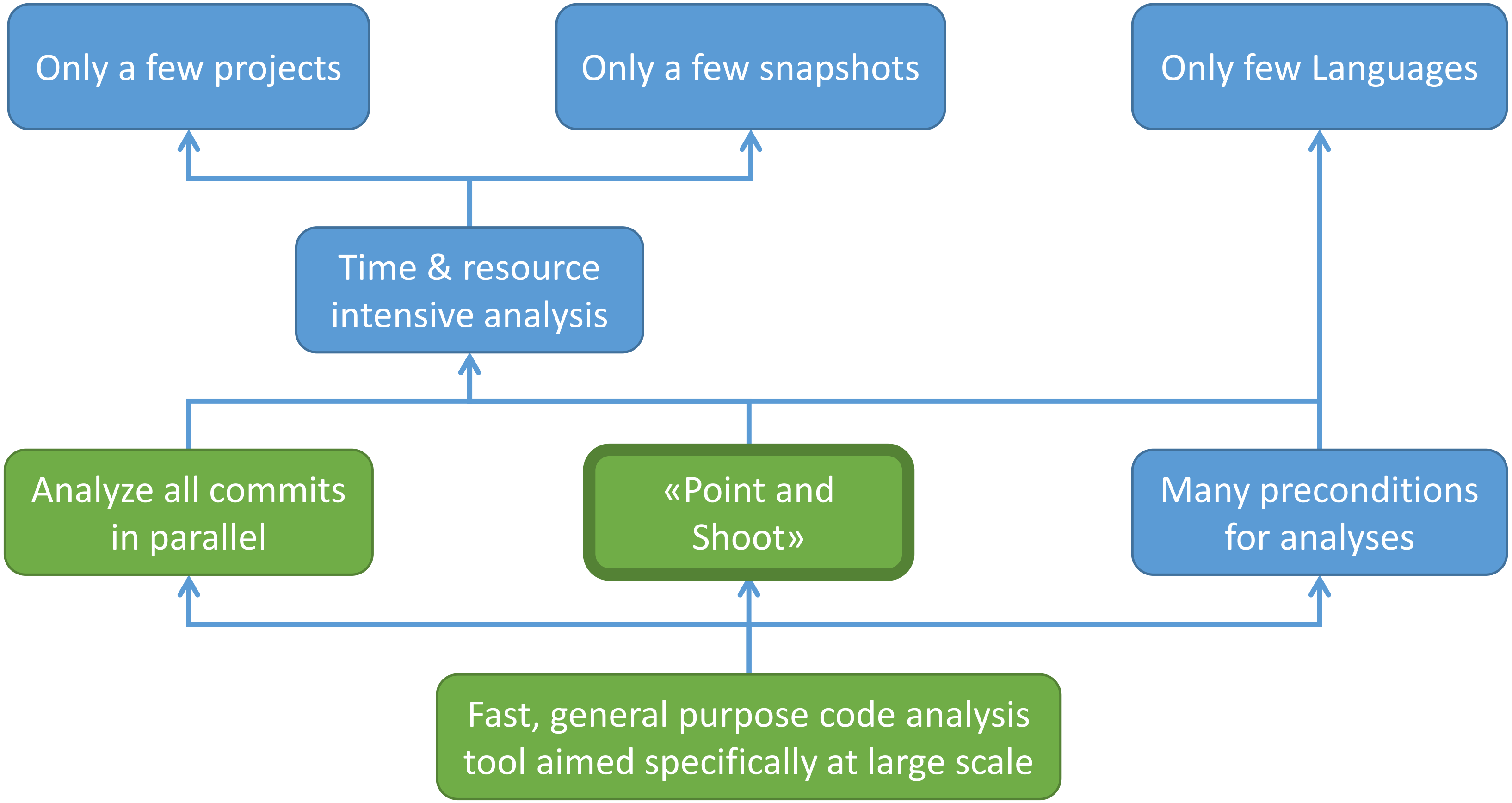
Analyze all commits in parallel

Manual work

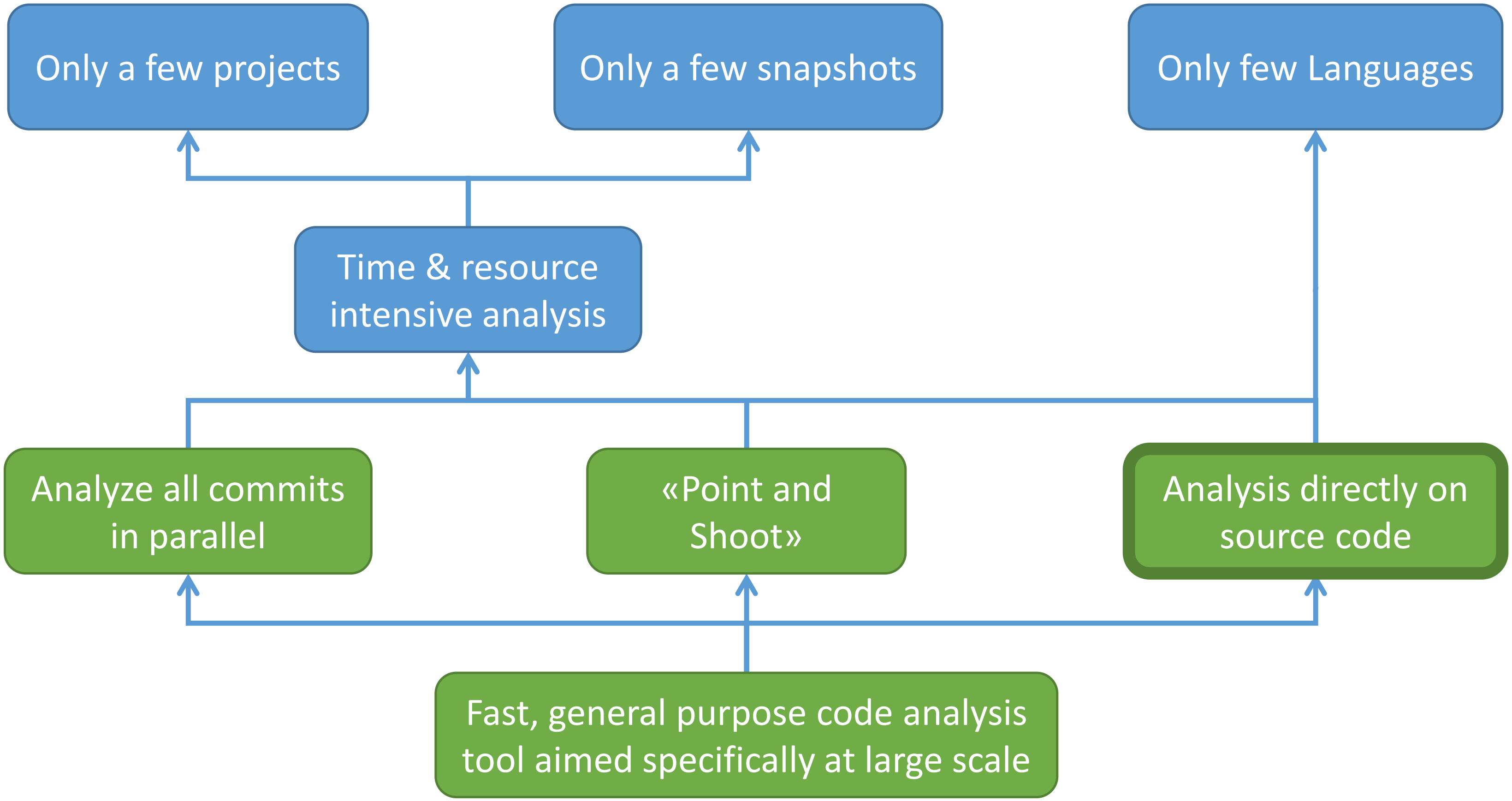
Many preconditions for analyses

Fast, general purpose code analysis tool aimed specifically at large scale

Rapid Analysis using LISA


Only a few projects
Only a few snapshots
Only few Languages
Time & resource intensive analysis
Analyze all commits in parallel
«Point and Shoot»
Many preconditions for analyses
Fast, general purpose code analysis tool aimed specifically at large scale

Rapid Analysis using LISA



Only a few projects

Only a few snapshots

Only few Languages

Time & resource intensive analysis

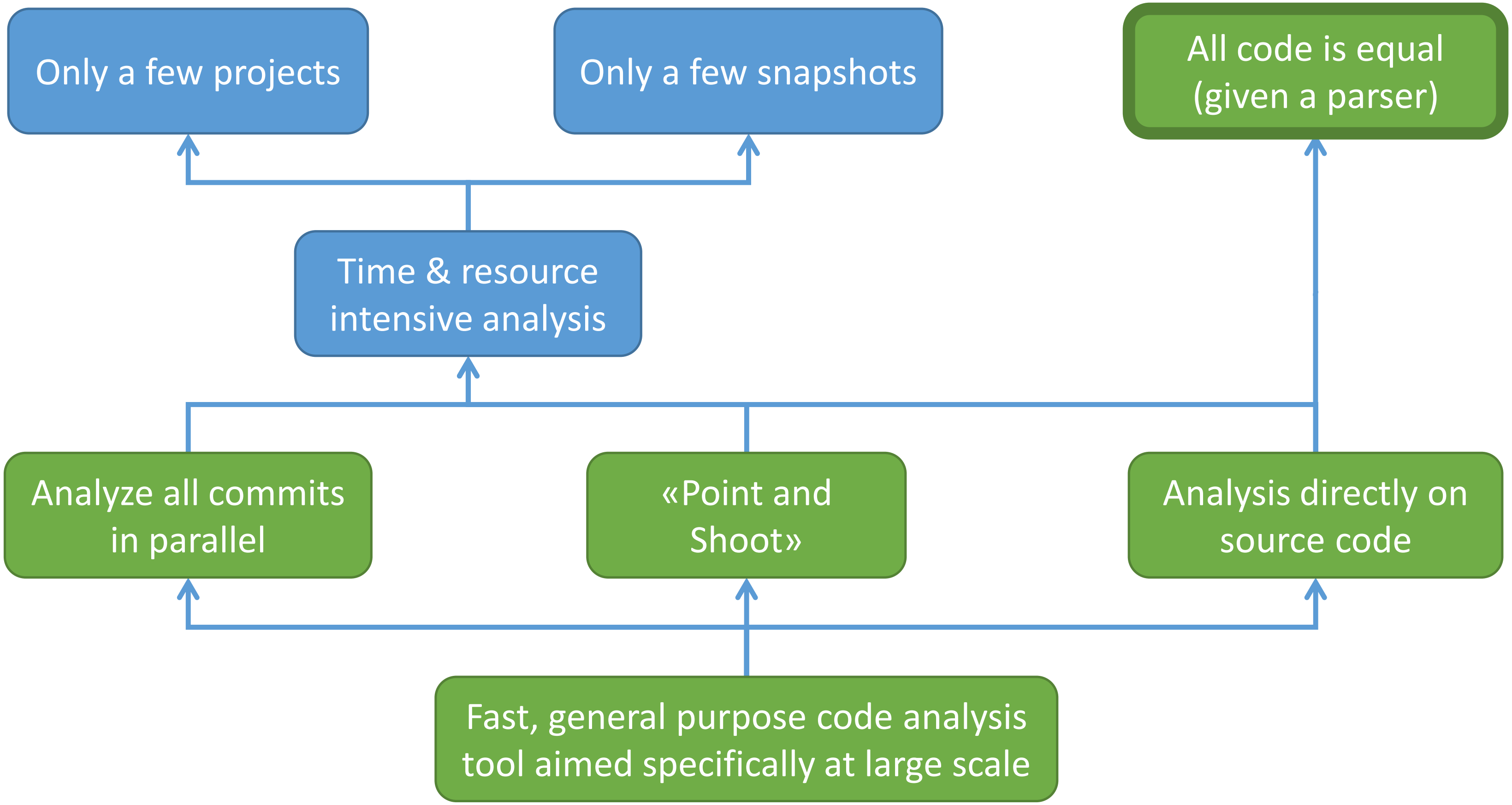
Analyze all commits in parallel

«Point and Shoot»

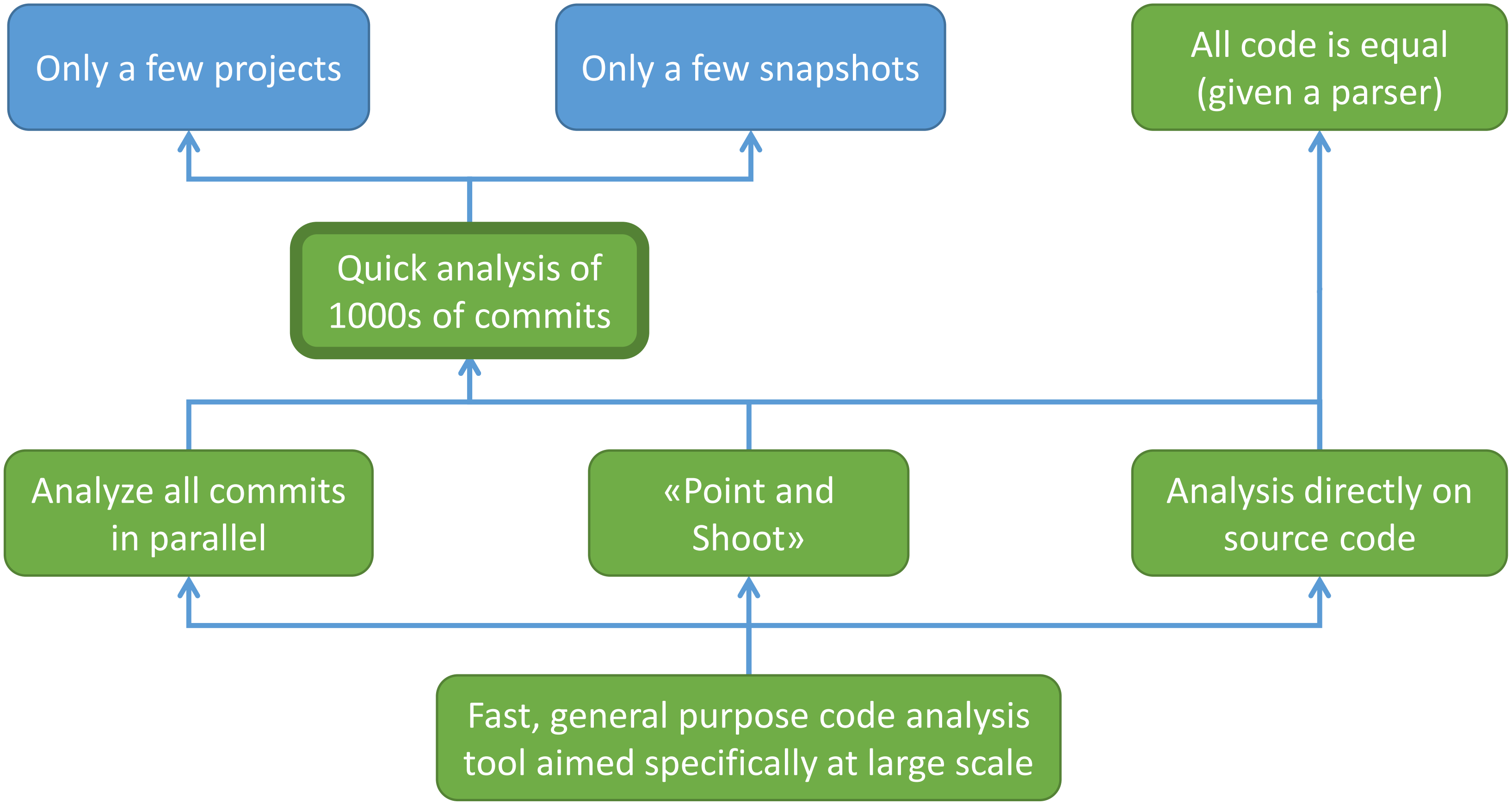
Analysis directly on source code

Fast, general purpose code analysis tool aimed specifically at large scale

Rapid Analysis using LISA


Only a few projects
Only a few snapshots
All code is equal (given a parser)
Time & resource intensive analysis
Analyze all commits in parallel
«Point and Shoot»
Analysis directly on source code
Fast, general purpose code analysis tool aimed specifically at large scale

Rapid Analysis using LISA



Only a few projects

Only a few snapshots

All code is equal (given a parser)

Quick analysis of 1000s of commits

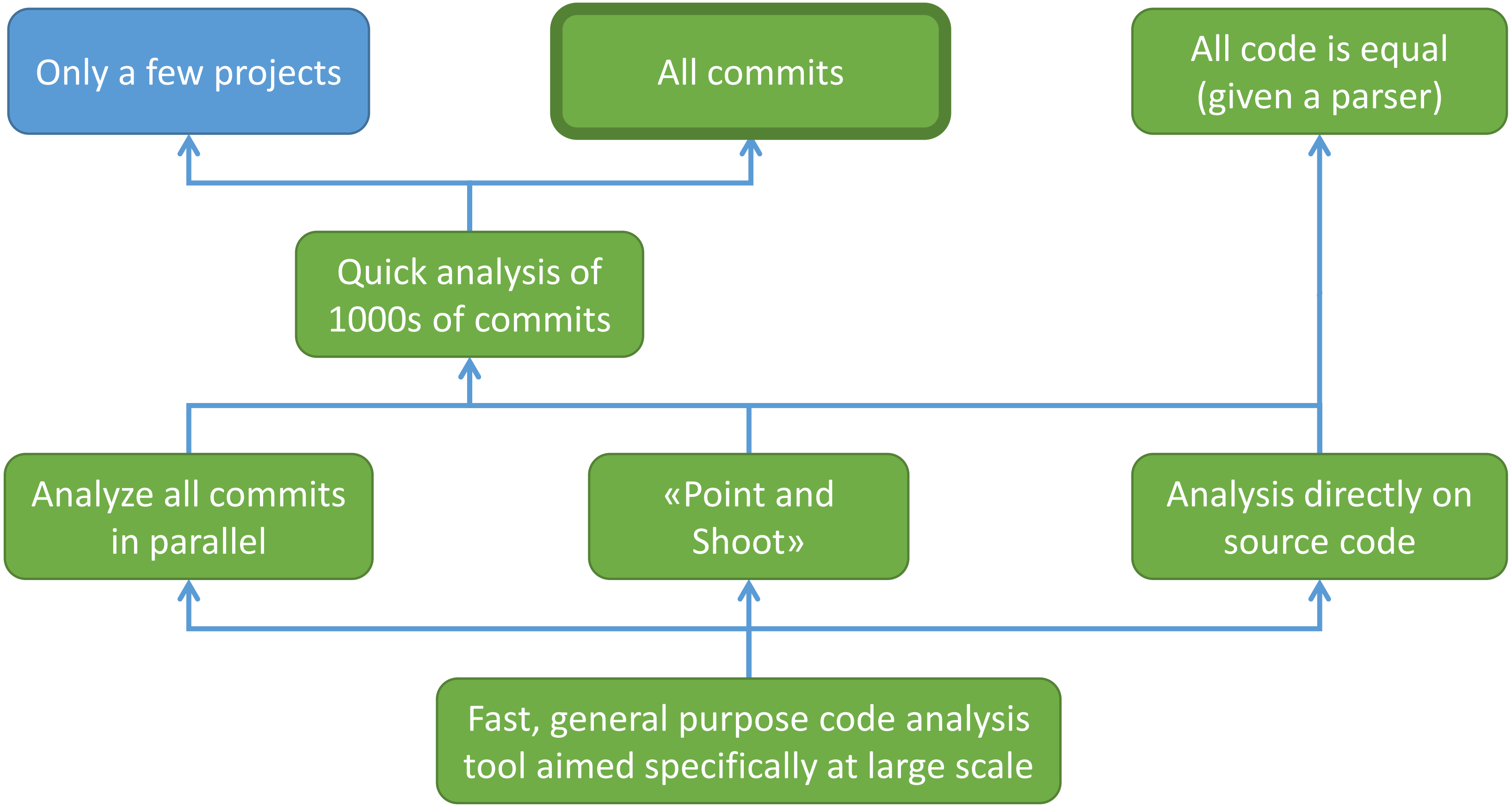
Analyze all commits in parallel

«Point and Shoot»

Analysis directly on source code

Fast, general purpose code analysis tool aimed specifically at large scale

Rapid Analysis using LISA



Only a few projects

All commits

All code is equal (given a parser)

Quick analysis of 1000s of commits

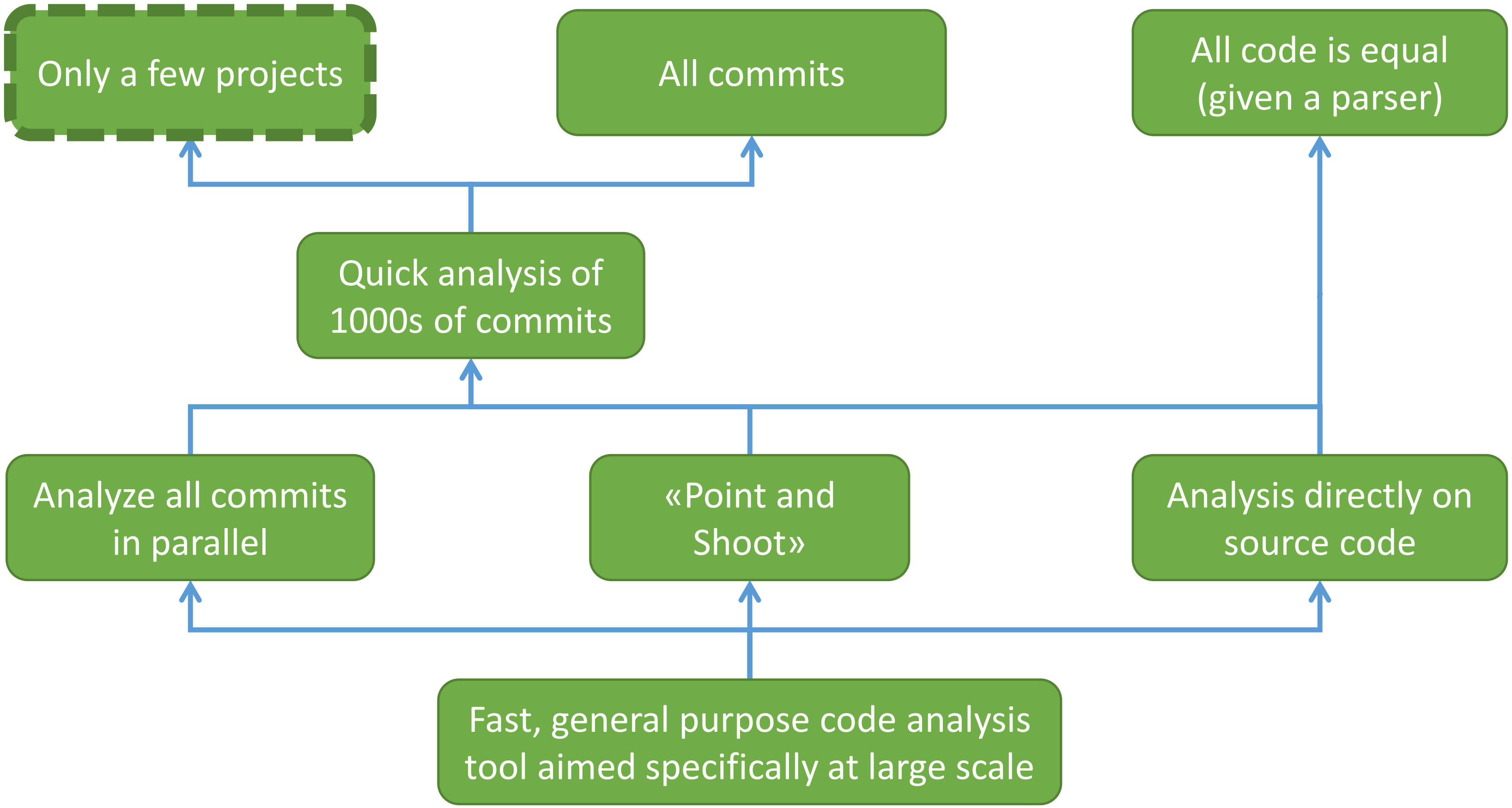
Analyze all commits in parallel

«Point and Shoot»

Analysis directly on source code

Fast, general purpose code analysis tool aimed specifically at large scale

Rapid Analysis using LISA



Only a few projects

All commits

All code is equal (given a parser)

Quick analysis of 1000s of commits

Analyze all commits in parallel

«Point and Shoot»

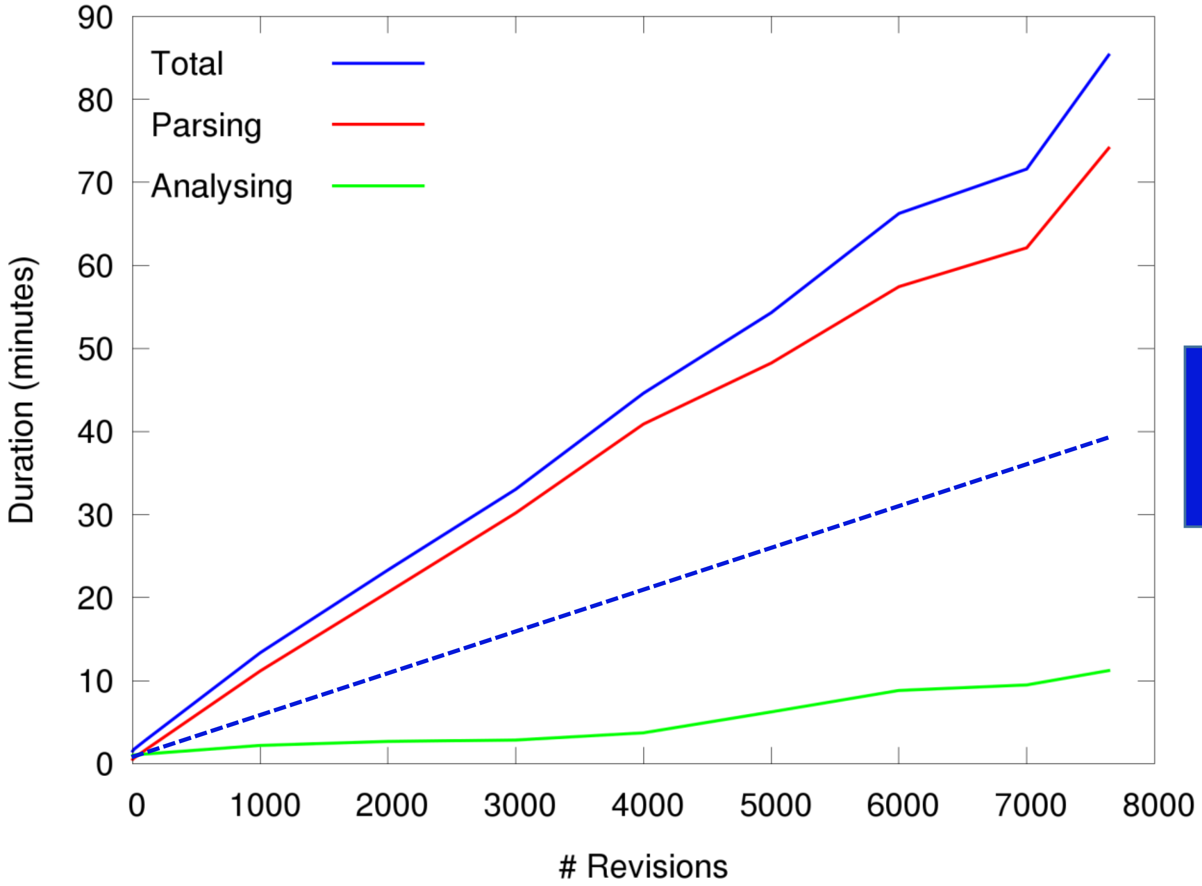
Analysis directly on source code

Fast, general purpose code analysis tool aimed specifically at large scale

University of Zurich UZH

s.e.a.l. software evolution & architecture lab

# of Commits: Linear Scaling



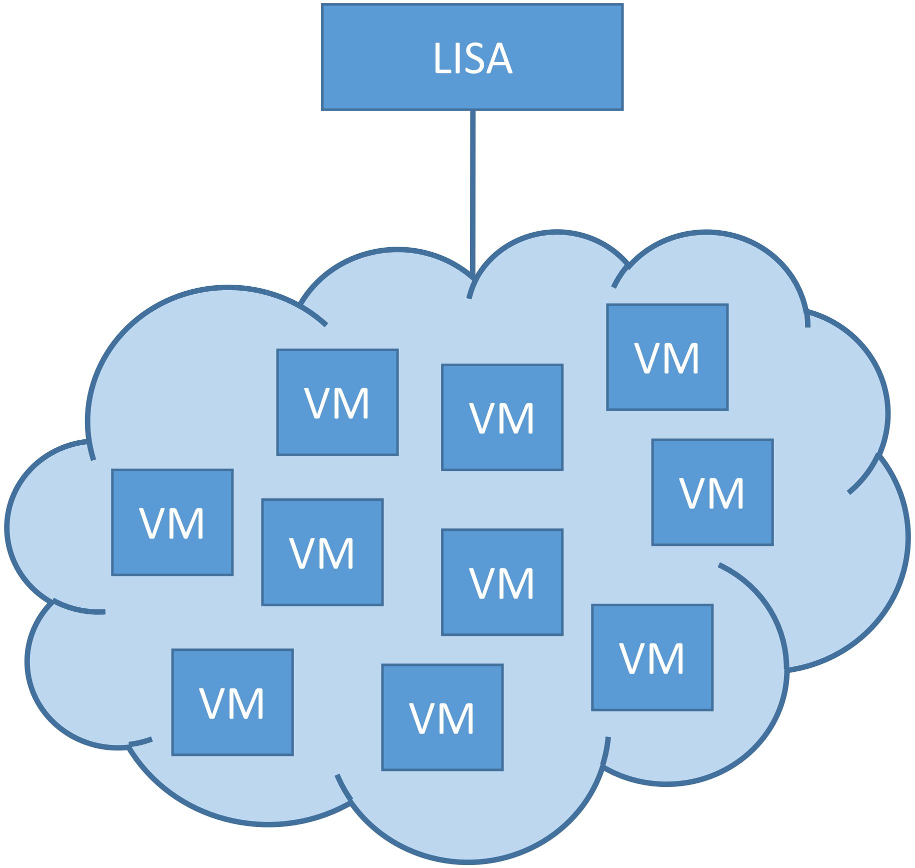
March Update (Total)

Multi-Project Parallelization

LISA

AspectJ:
  7642 commits, 440k LOC
Requirements:
  20 GB memory
  45 min on 4 cores

Multi-Project Parallelization

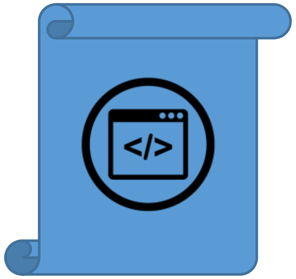


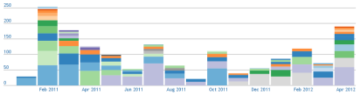
AspectJ:
 7642 commits, 440k LOC
Requirements:
 20 GB memory
 45 min on 4 cores

Parallelization scenario:
 10x Amazon EC 2 r3.8xlarge
 10x 244GB memory
 10x 32 cores
Potential to analyze 160 AspectJ-sized projects per hour

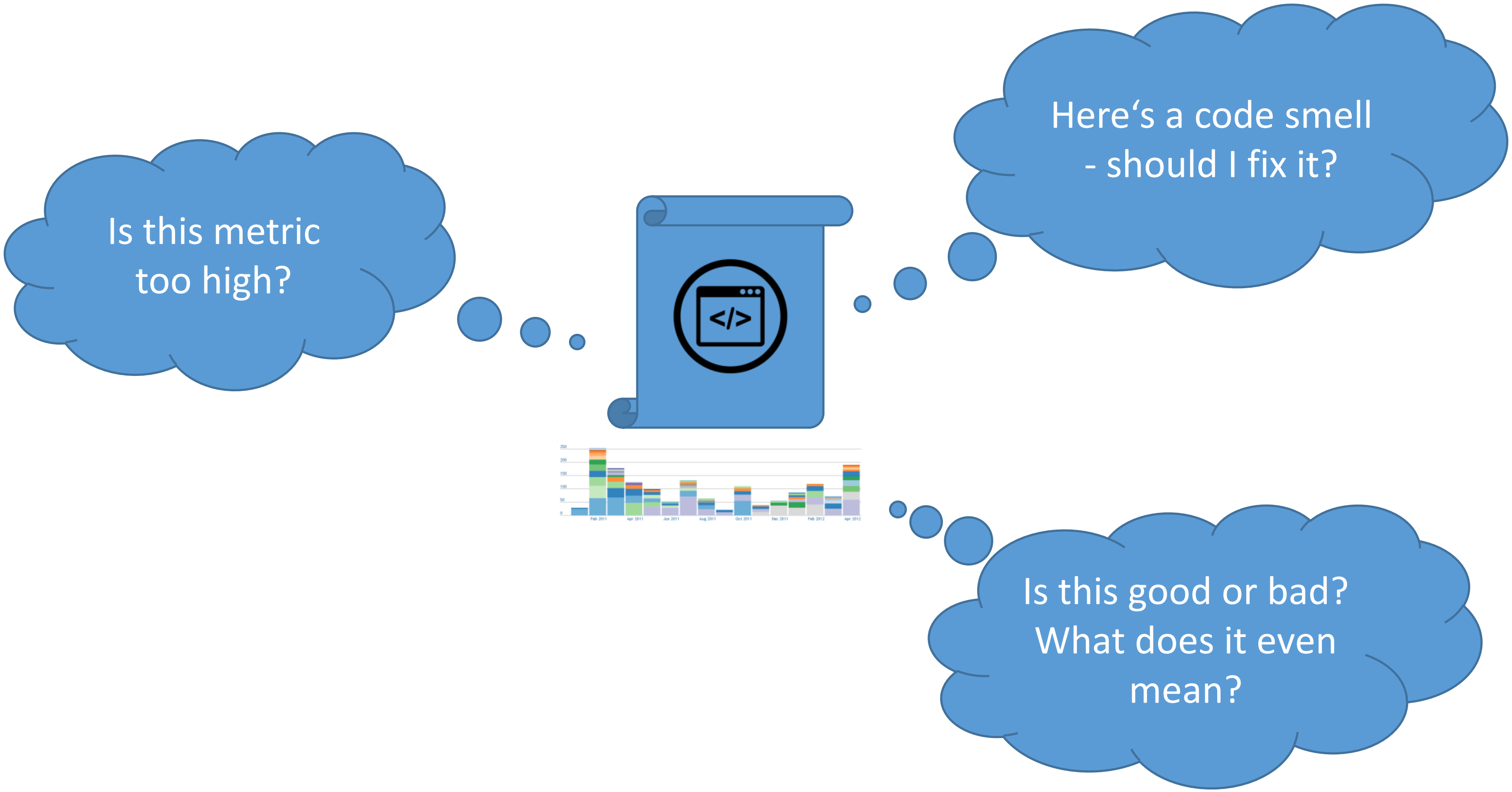
(Most projects on git-hub are much smaller)

Benchmarking





Benchmarking



Is this metric too high?

Here‘s a code smell - should I fix it?

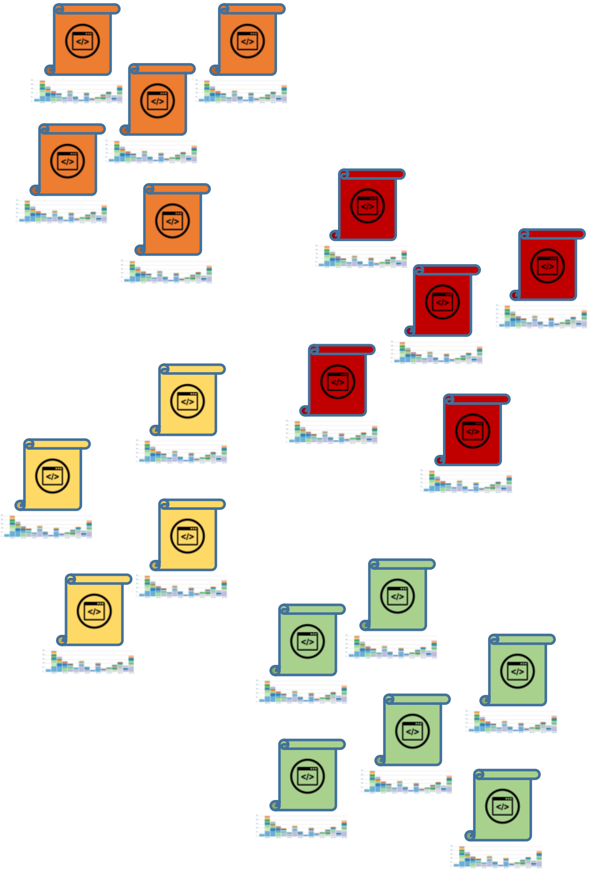
Is this good or bad? What does it even mean?

Benchmarking

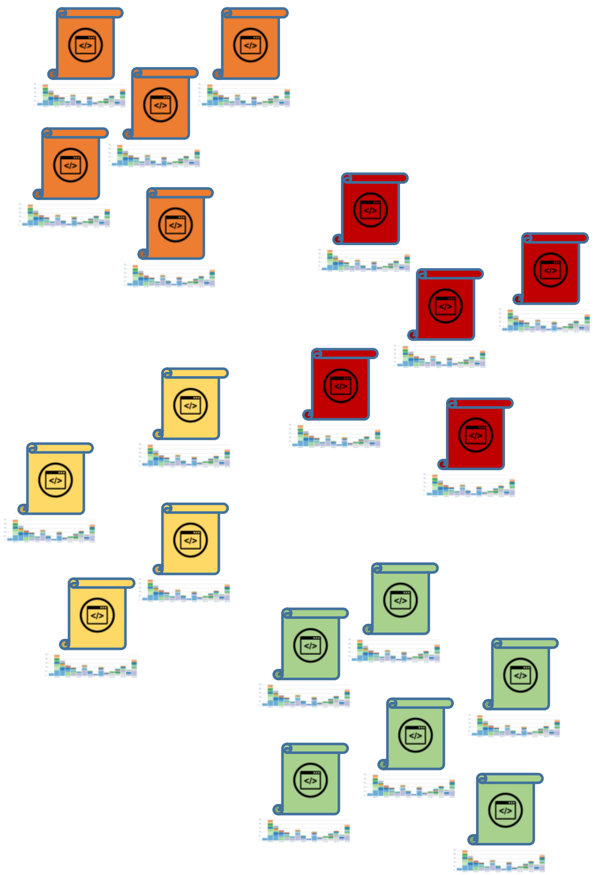
Benchmarking

Create reference points for orientation

Cluster & Compare

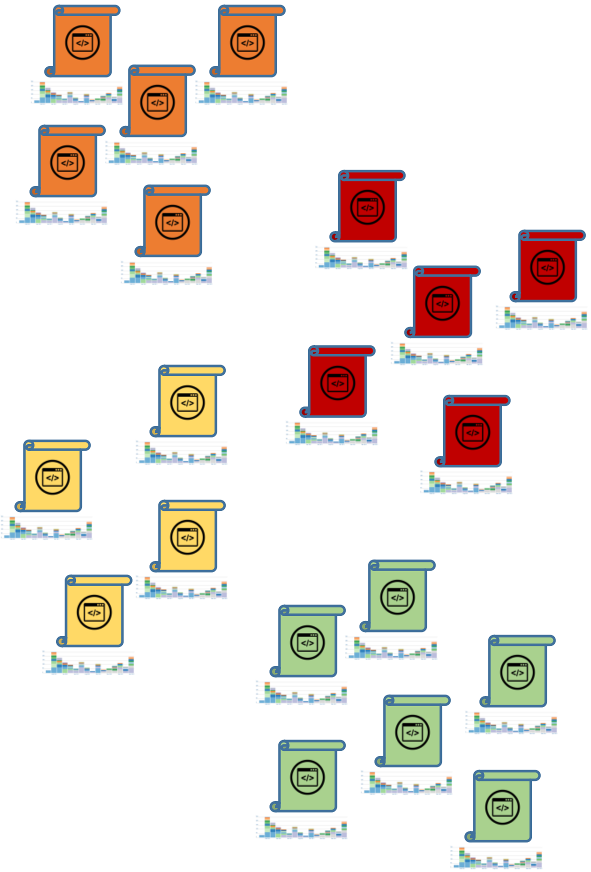


Cluster & Compare



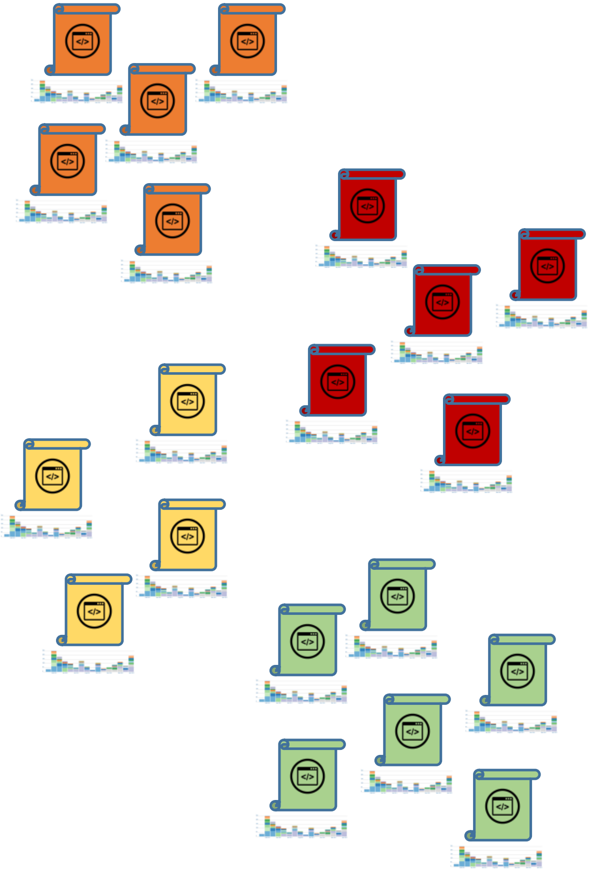
- Discover „phenotypes‟
    - By metric values

Cluster & Compare



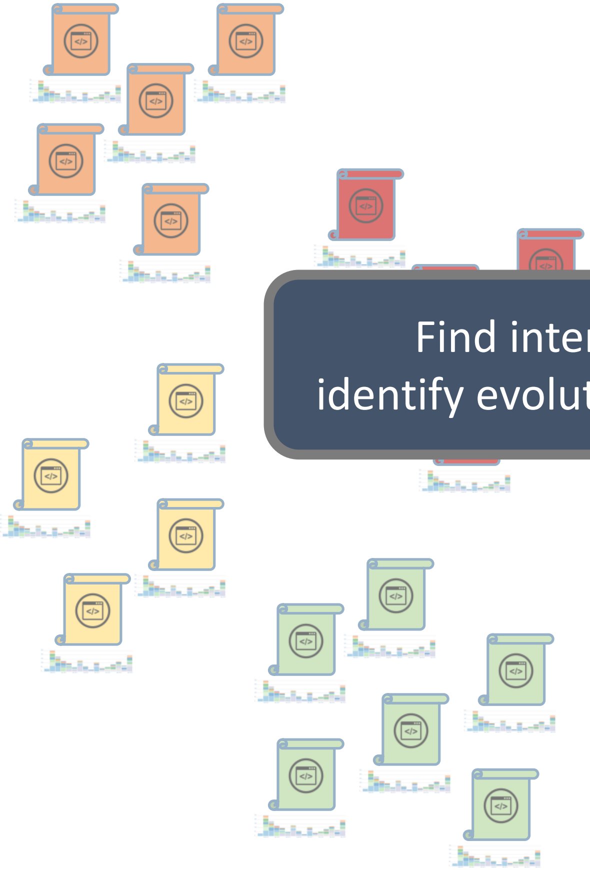
- Discover „phenotypes"
  - By metric values
  - By metric evolution over time

Cluster & Compare



- Discover „phenotypes“
  - By metric values
  - By metric evolution over time
- Compare programming languages / paradigms

Cluster & Compare

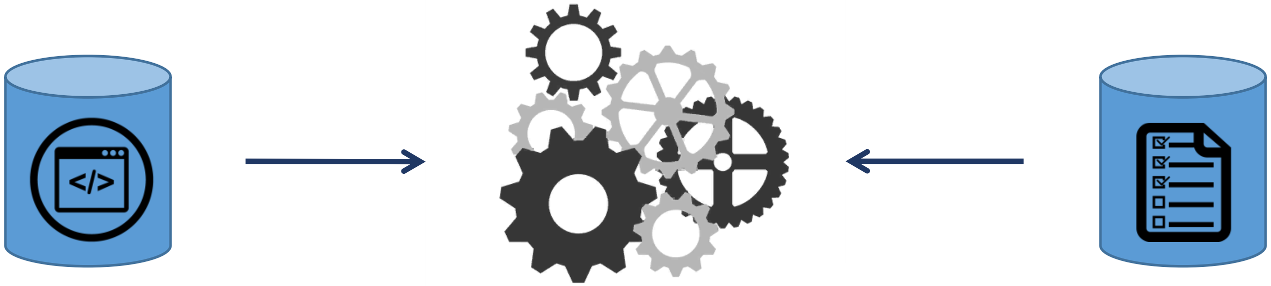


- Discover „phenotypes‟
    - By metric values
    - n over
      time

Find interesting projects for further study &
identify evolution patterns to find (un-)desirable ones

- Compare programming languages / paradigms

Machine Learning

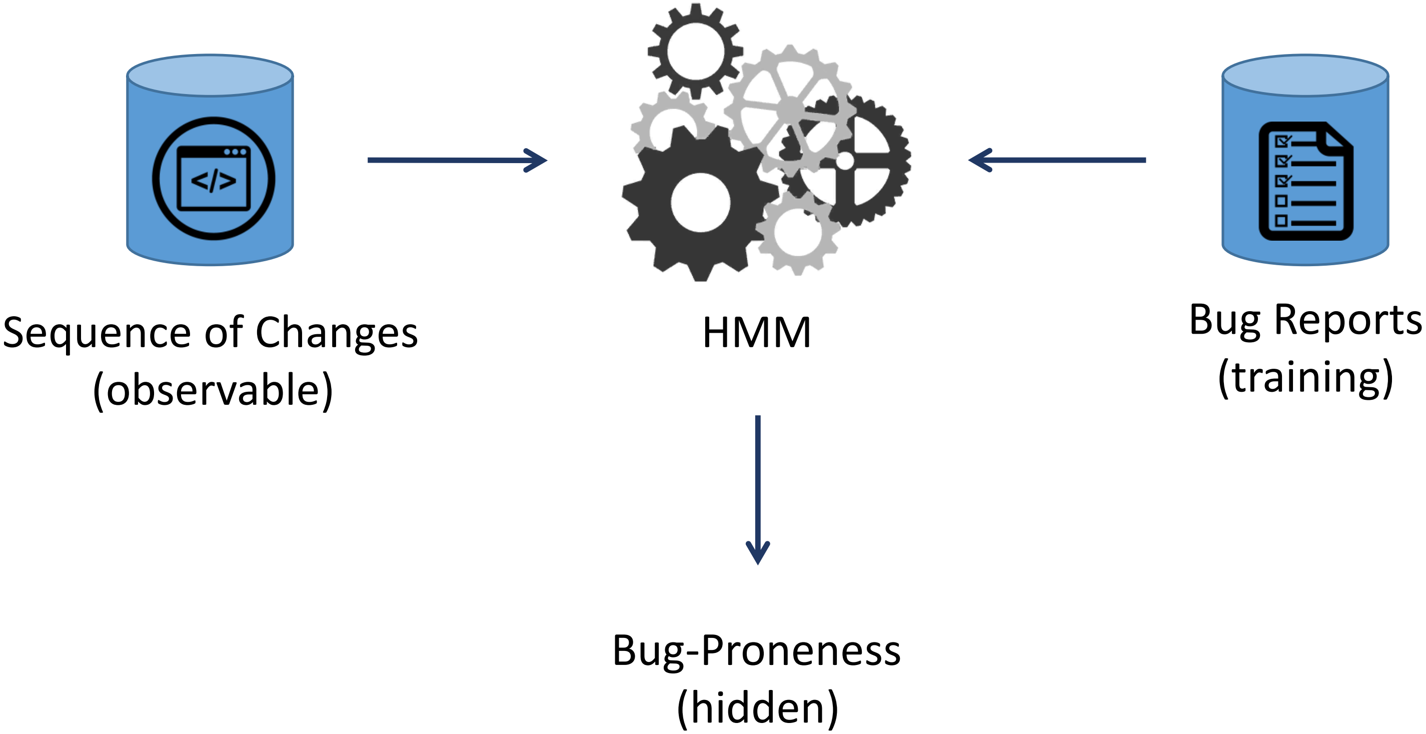


Sequence of Changes (observable)

HMM

Bug Reports (training)

Machine Learning



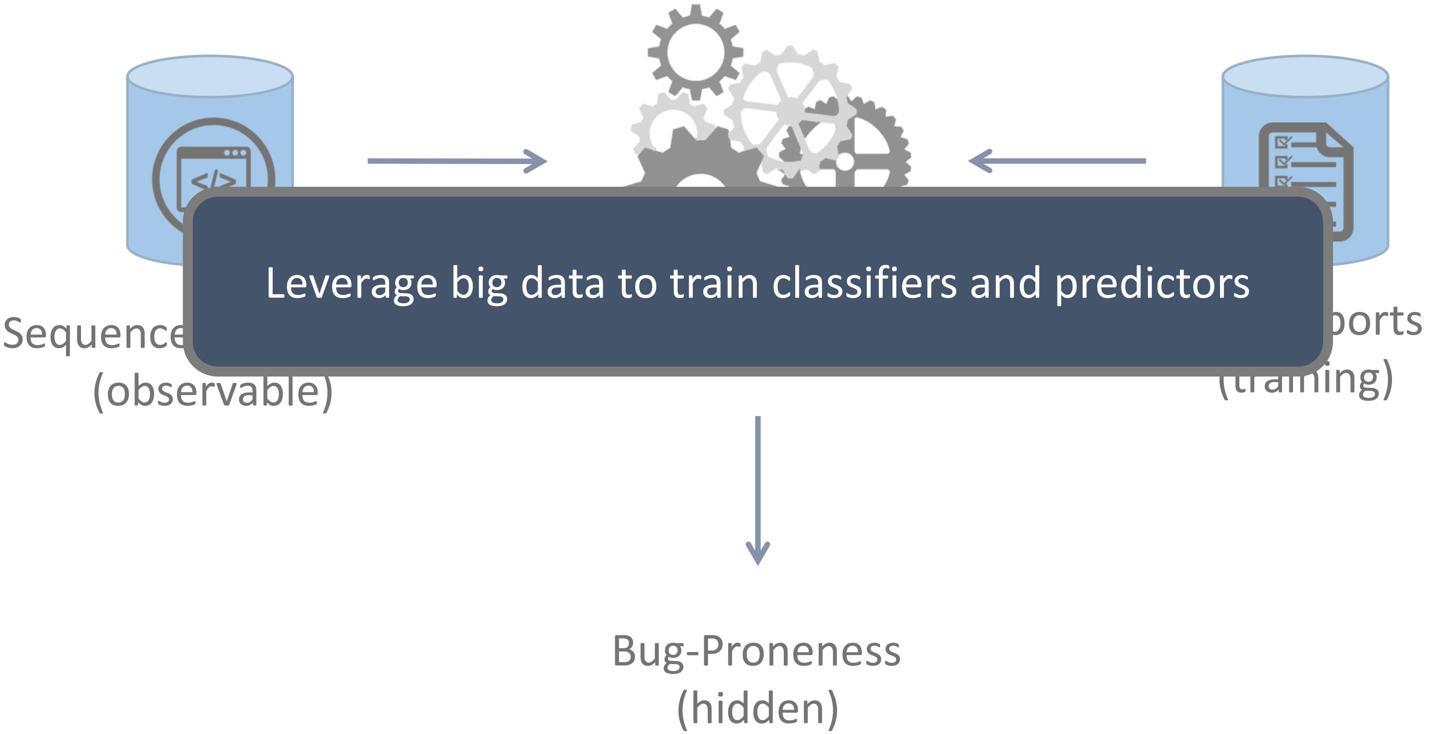
Sequence of Changes (observable)

HMM

Bug Reports (training)

Bug-Proneness (hidden)

Machine Learning



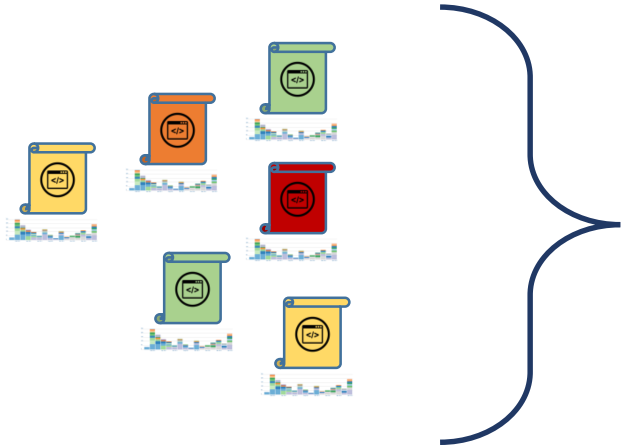
Leverage big data to train classifiers and predictors

Sequence
(observable)

ports
(training)

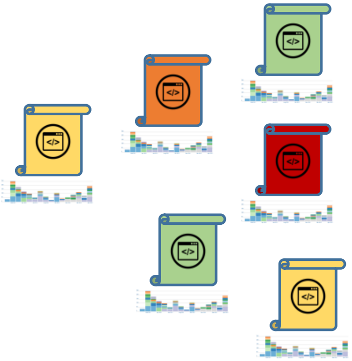
Bug-Proneness
(hidden)

Study Replication

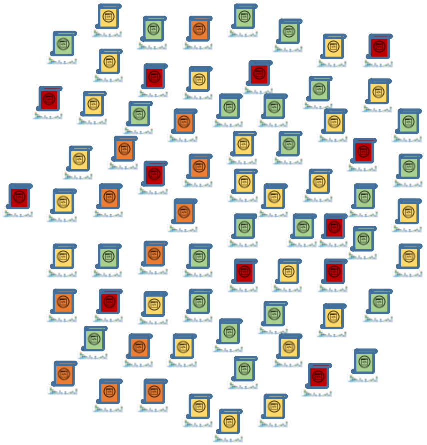


Conclusions X, Y, Z

Study Replication



Conclusions X, Y, Z

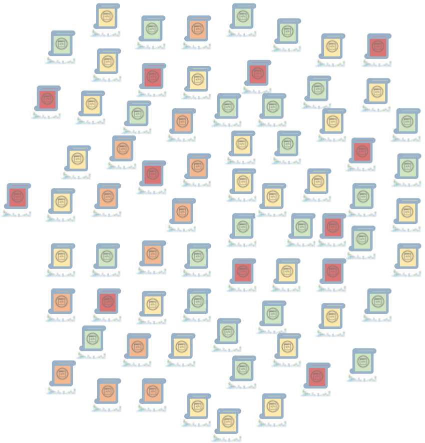


Replicable with 1000s of projects?

Study Replication

Conclusions X, Y, Z

Confirmation or rebuttal of existing assumptions



Replicable with 1000s of projects?