University of Zurich[UZH]

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

SANER'17

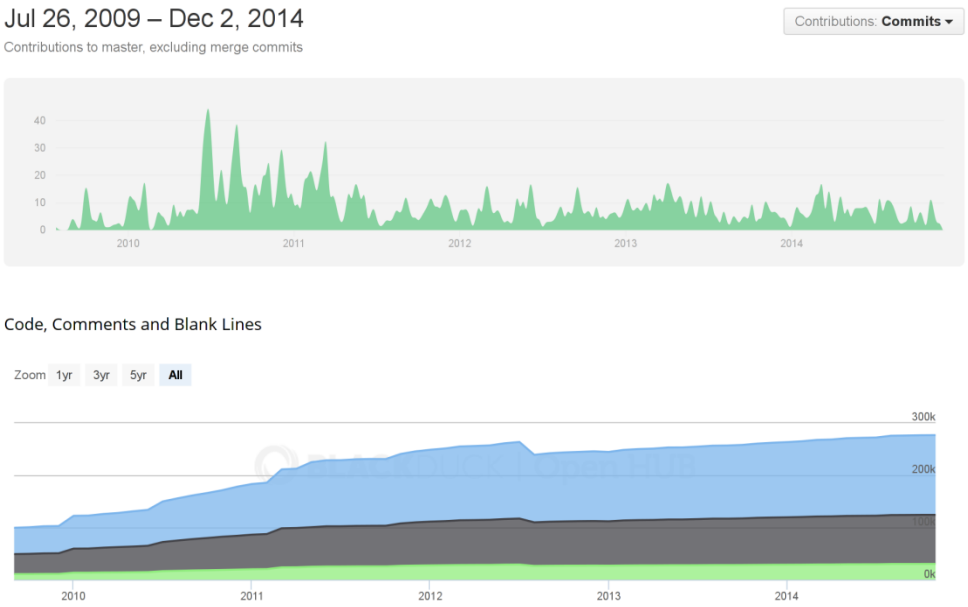
Klagenfurt, Austria

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

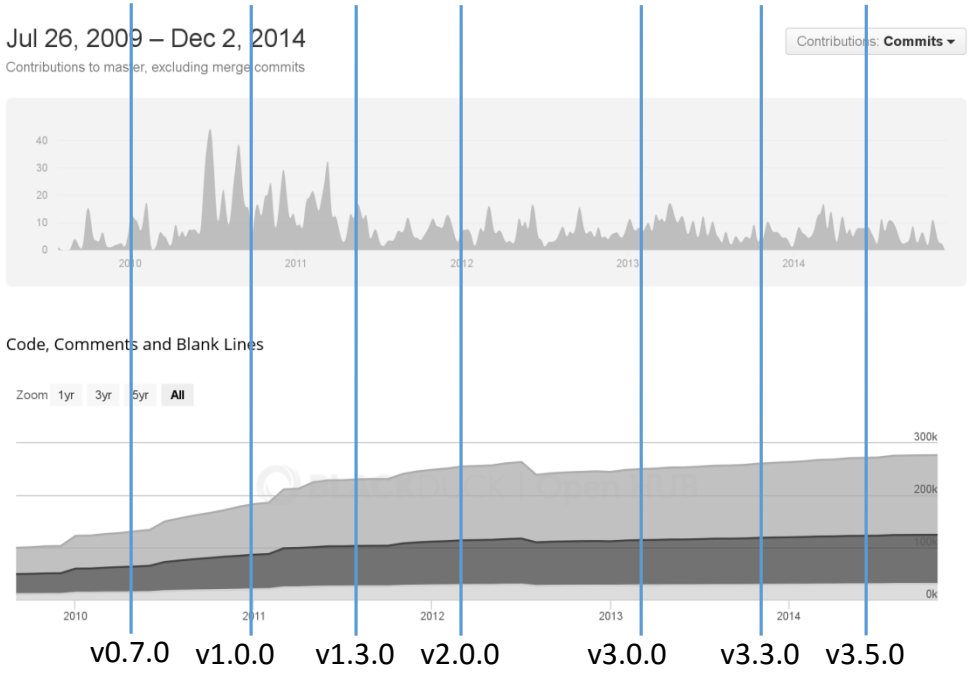
The Problem Domain

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



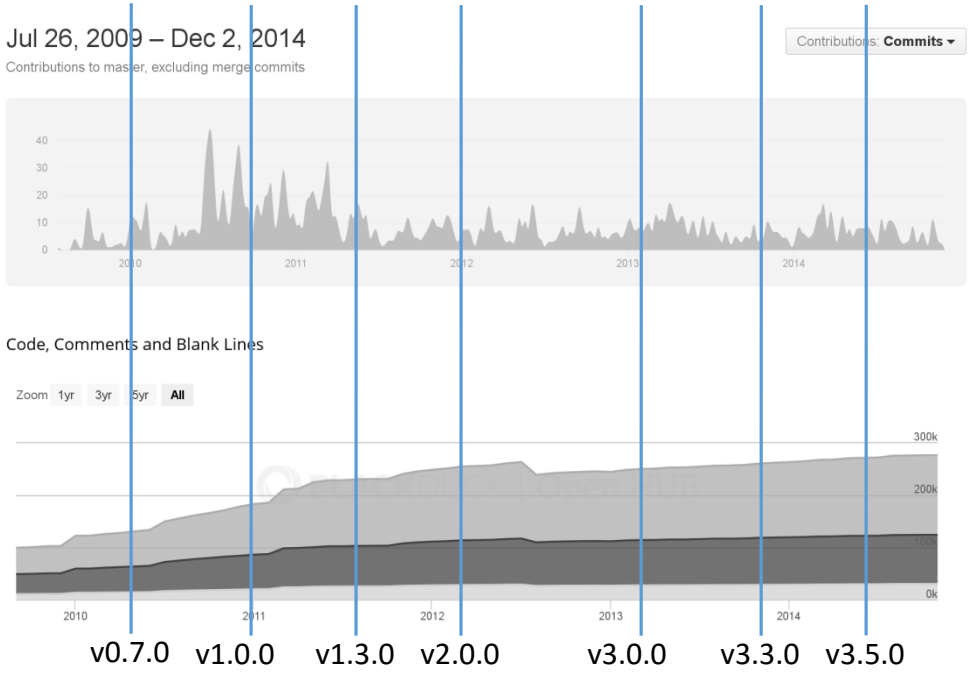
The Problem Domain

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

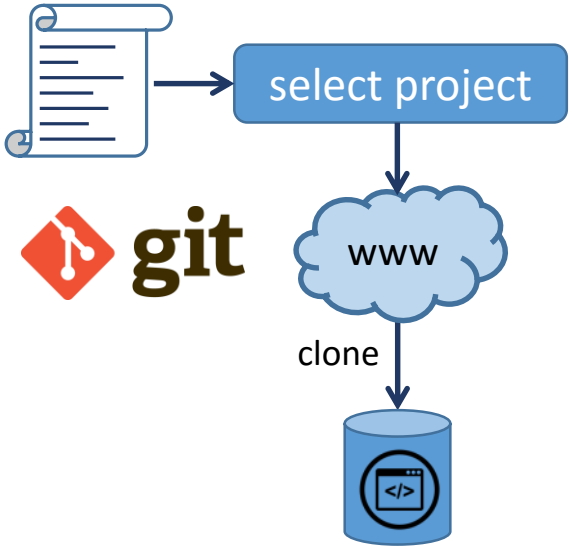


The Problem Domain

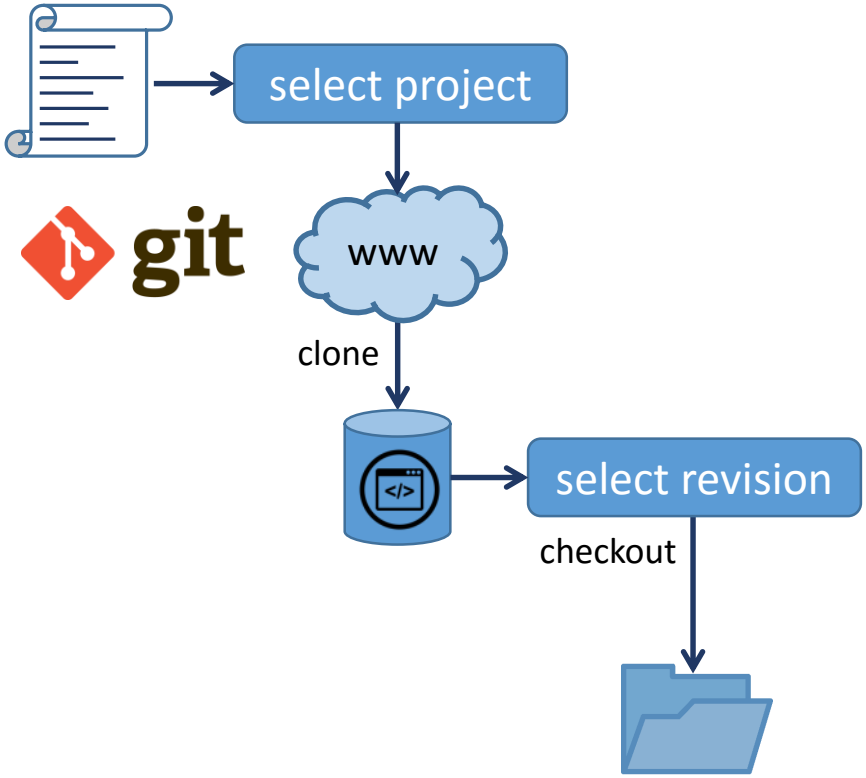
- Static analysis (e.g. #Attr., McCabe, coupling...)
- Many revisions, fine-grained historical data



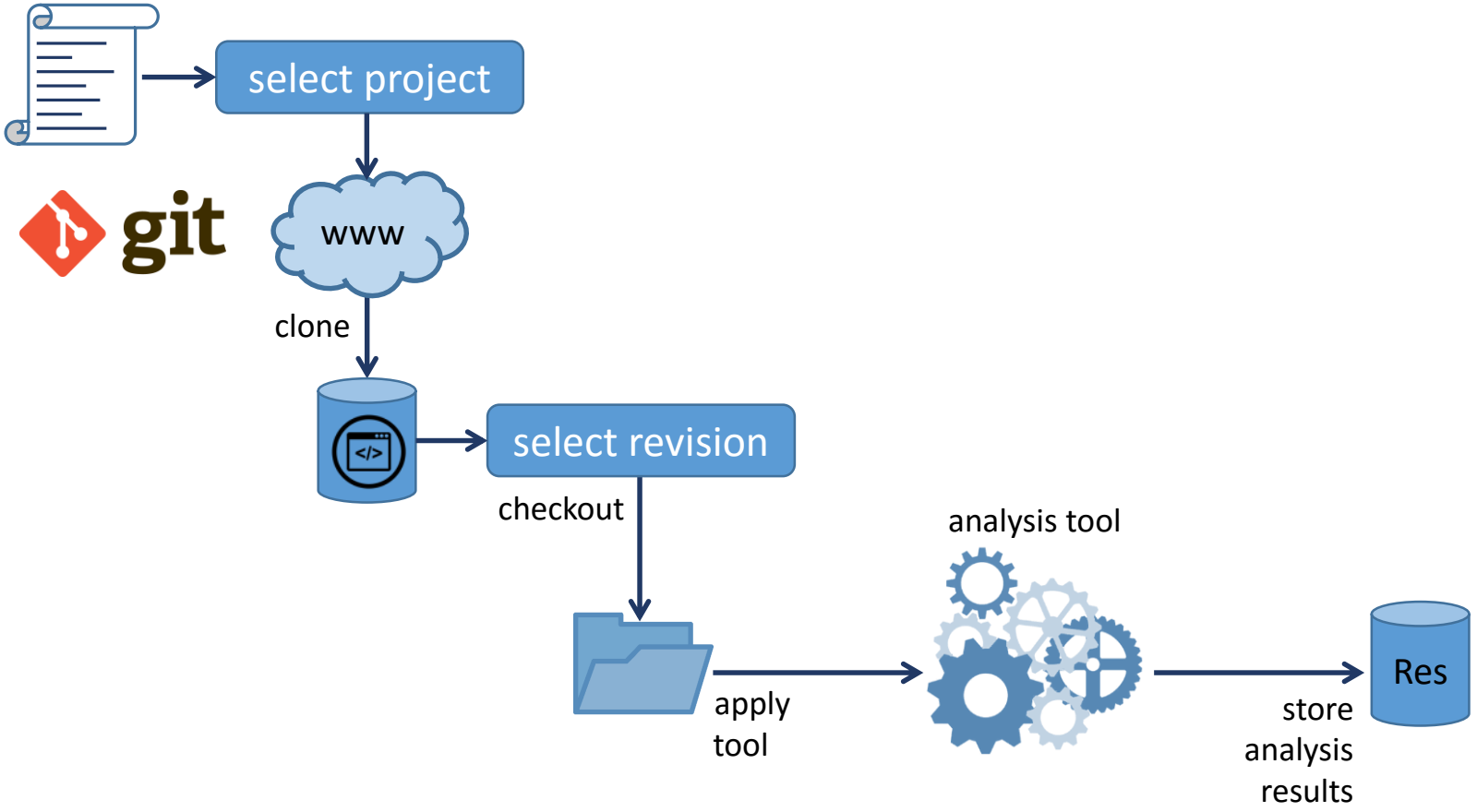
A Typical Analysis Process



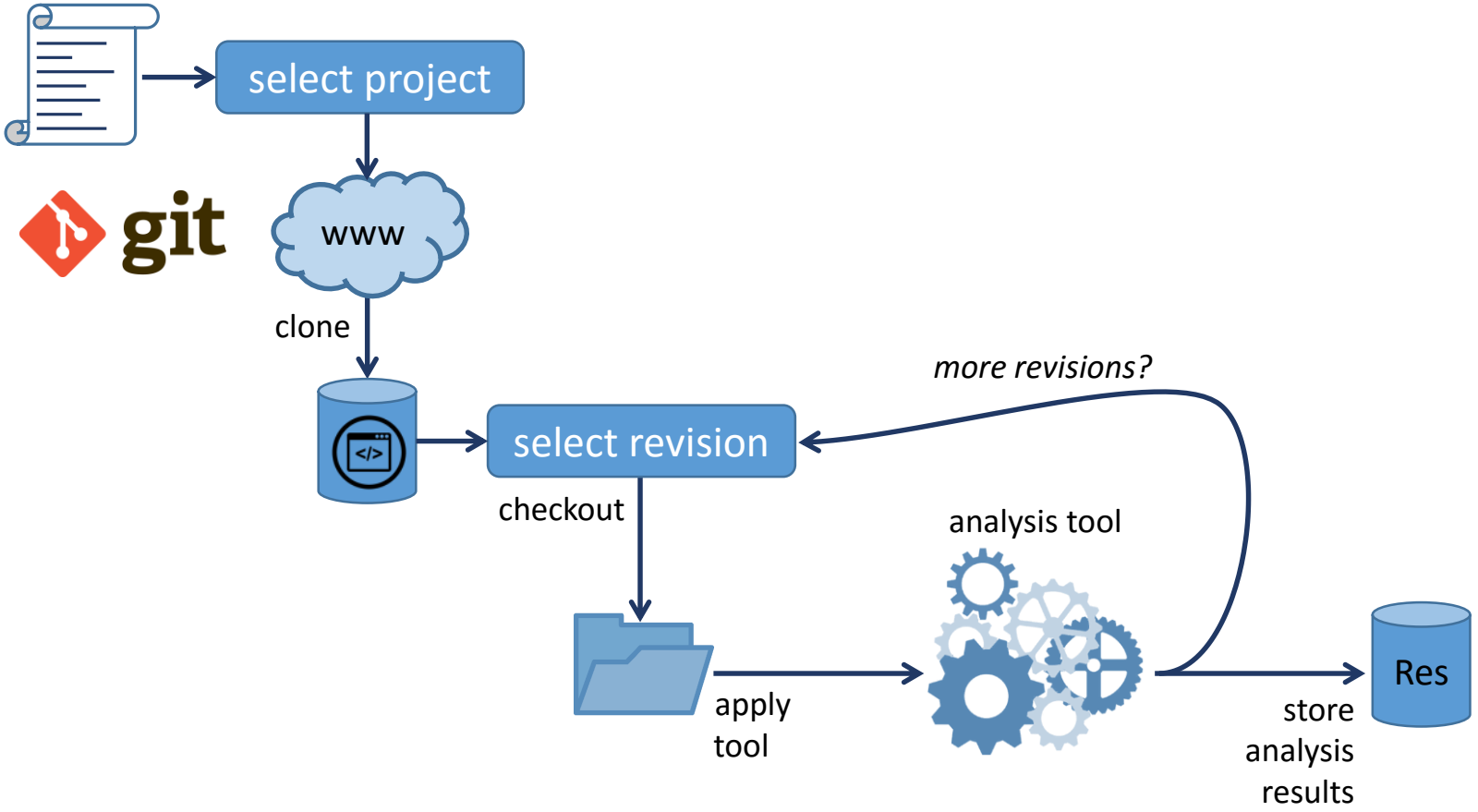
A Typical Analysis Process



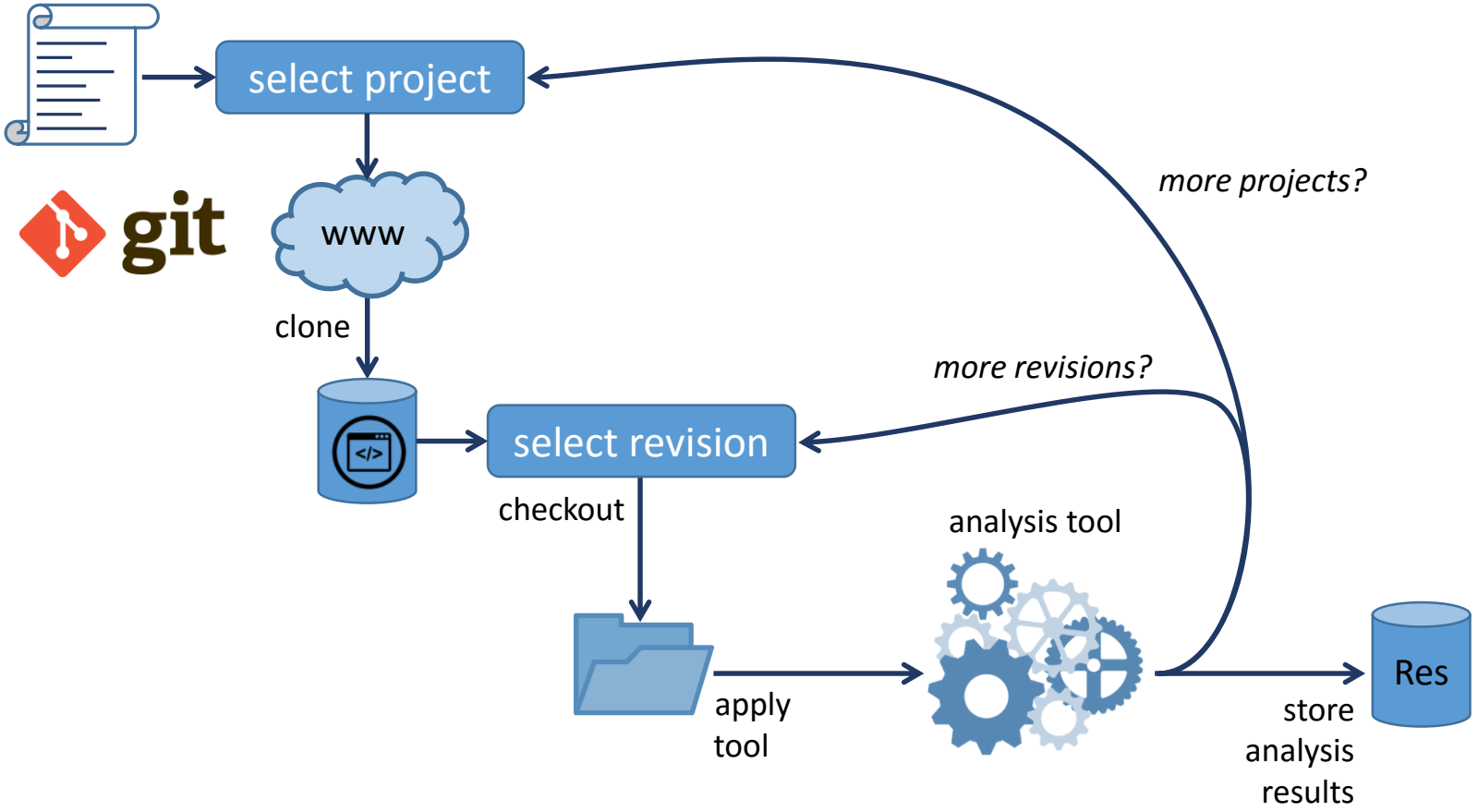
A Typical Analysis Process



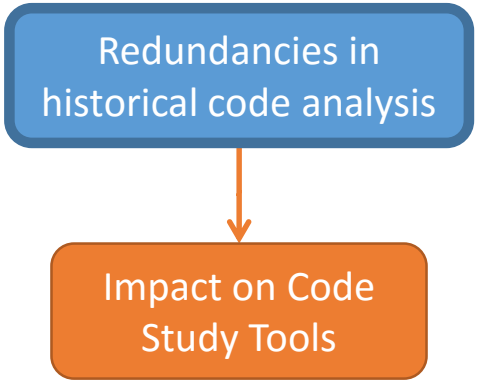
A Typical Analysis Process



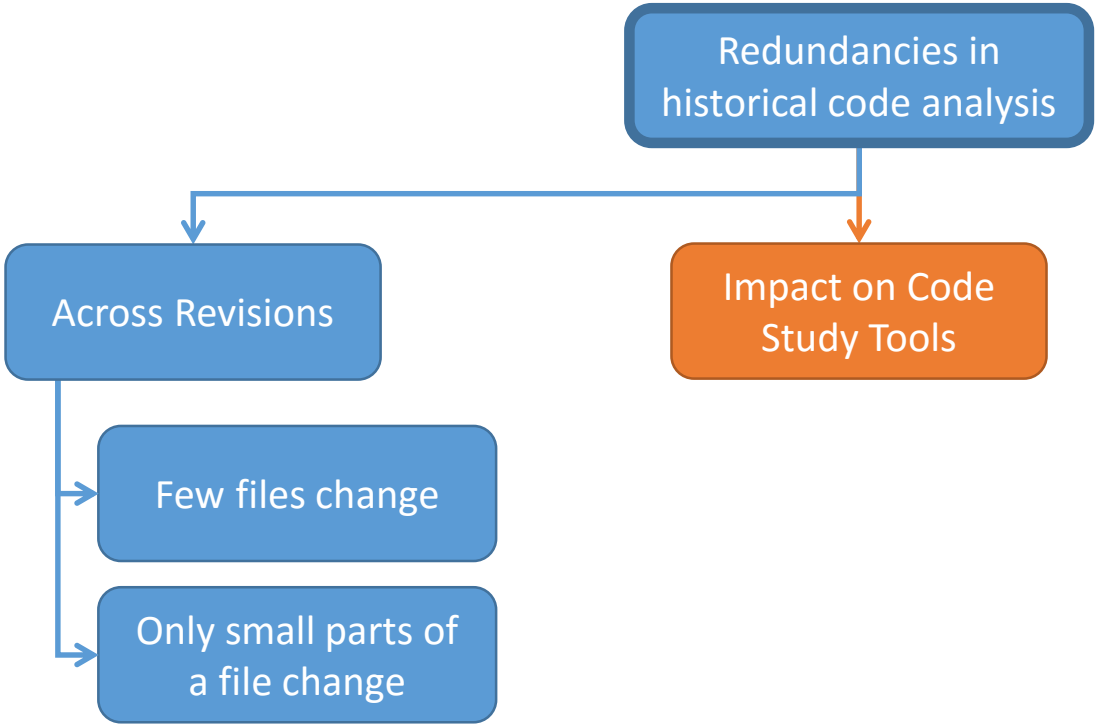
A Typical Analysis Process



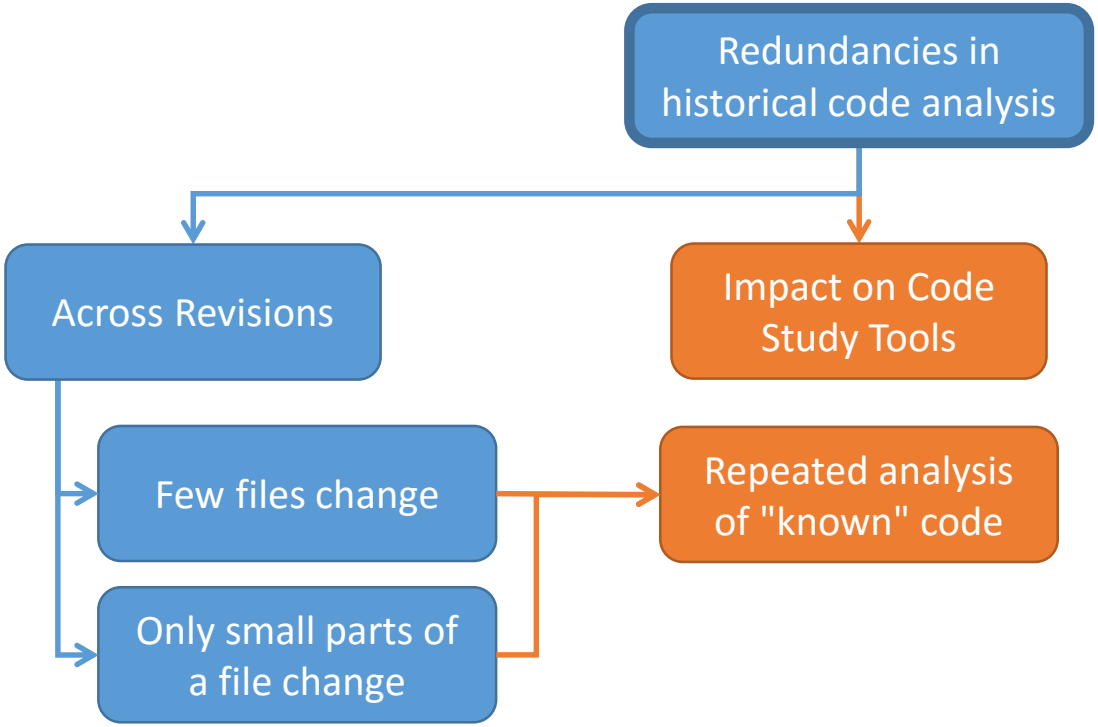
Redundancies all over...



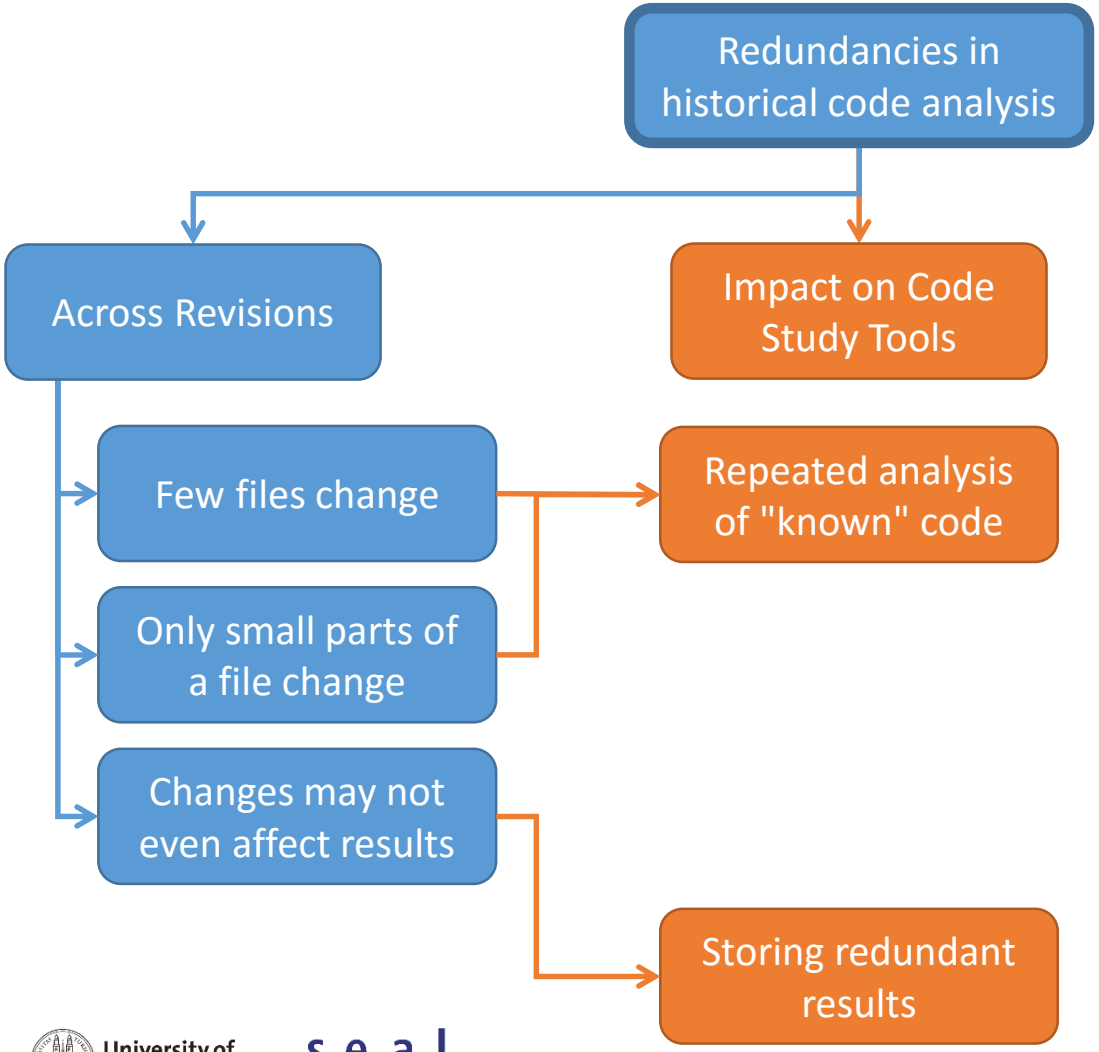
Redundancies all over...



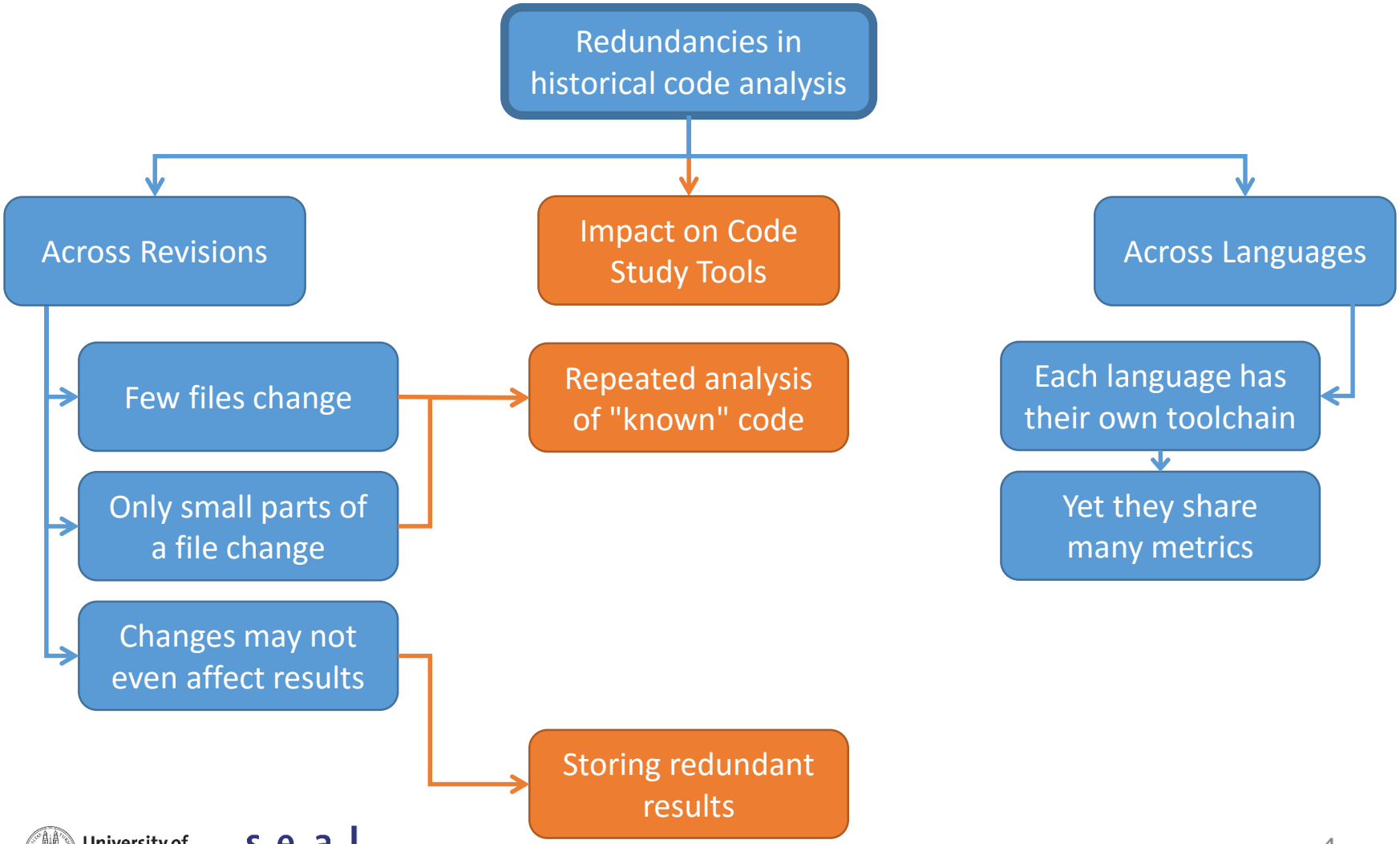
Redundancies all over...



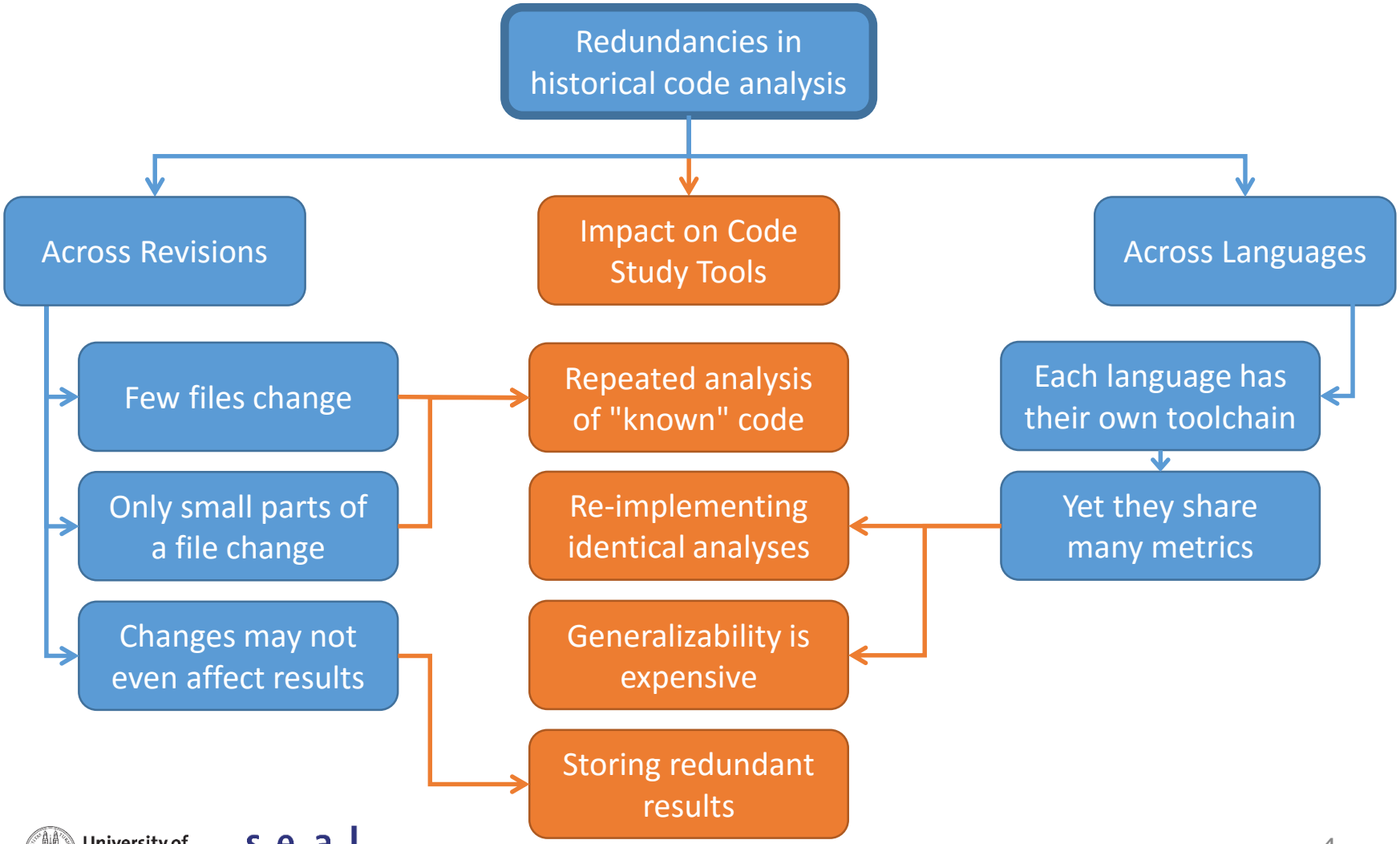
Redundancies all over...



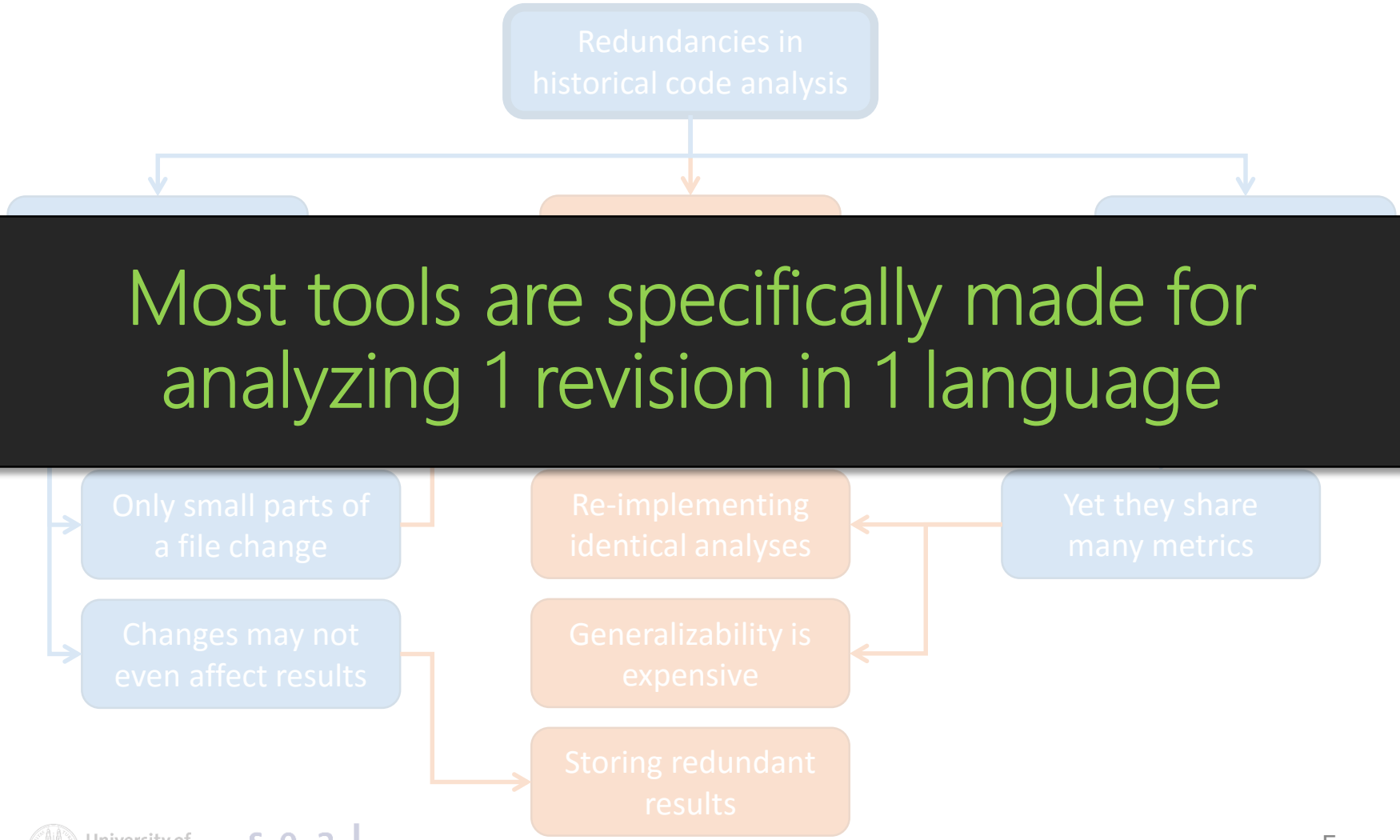
Redundancies all over...



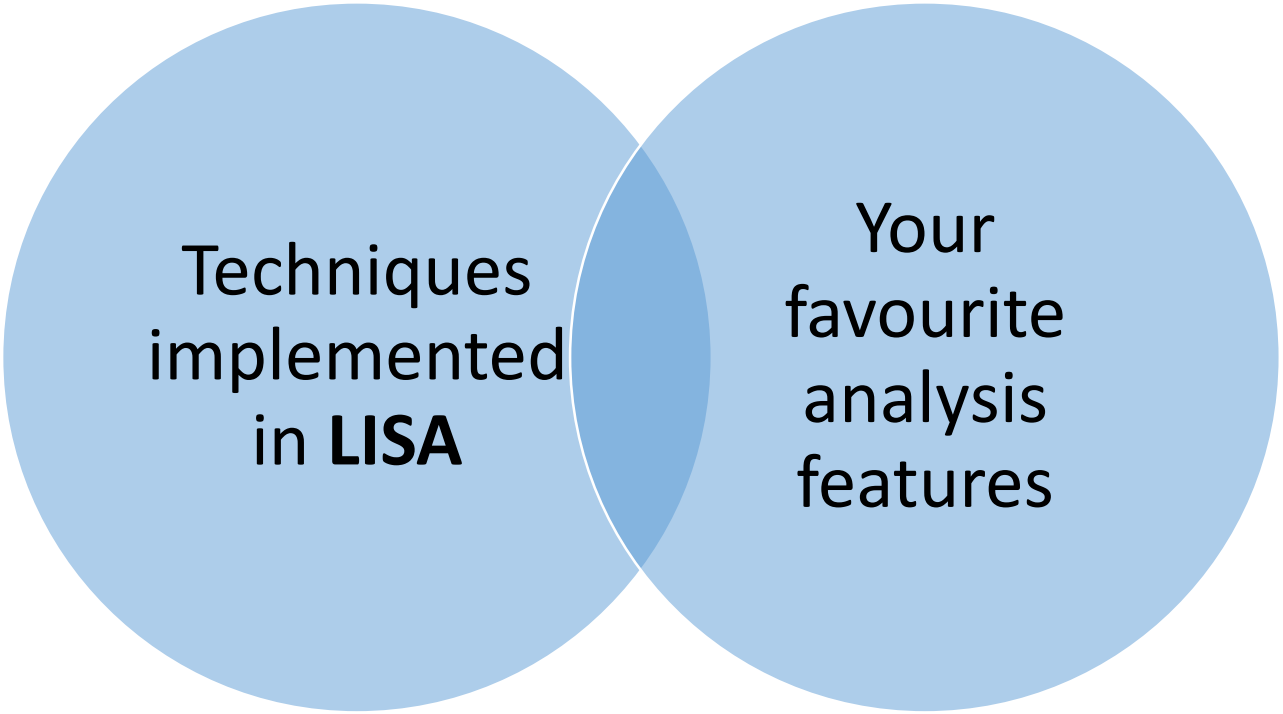
Redundancies all over...



Redundancies all over…



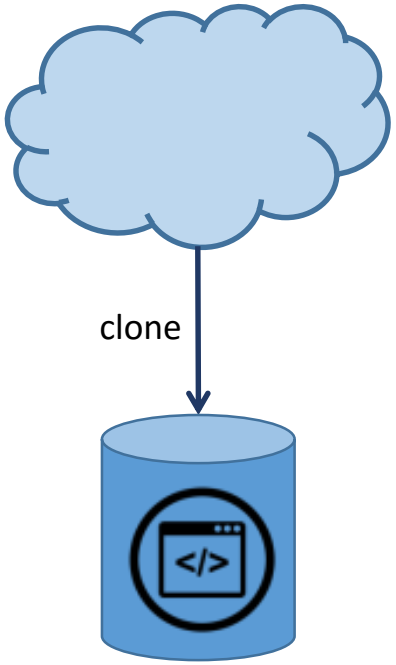
Important!



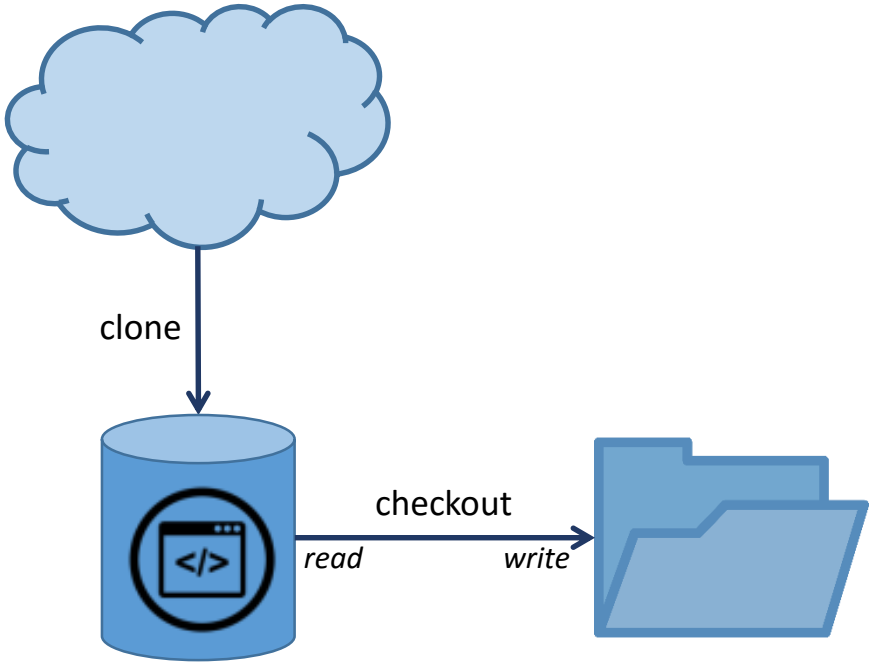
Pick what you like!

#1: Avoid Checkouts

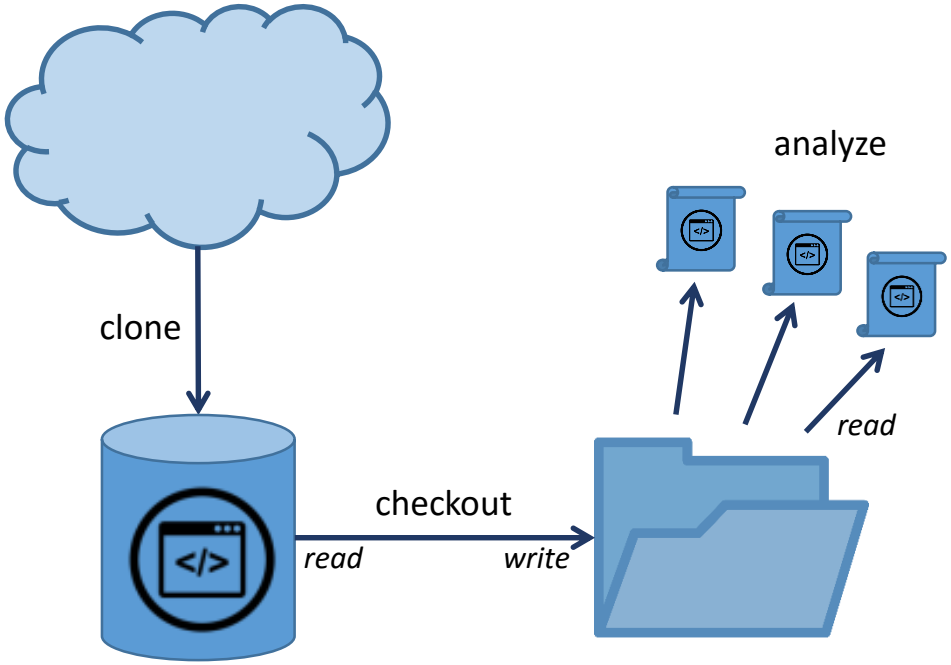
Avoid checkouts



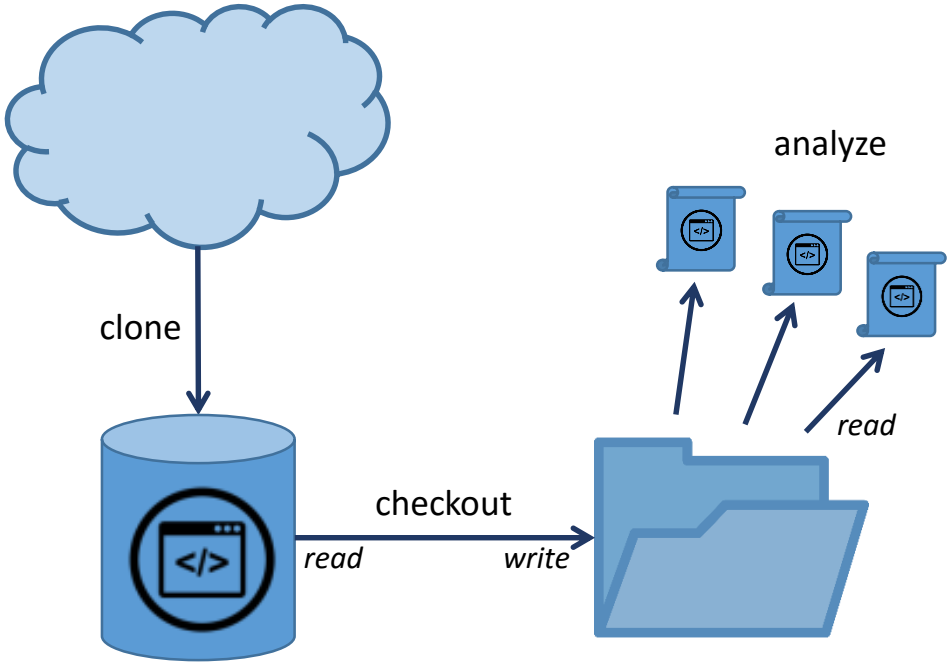
Avoid checkouts



Avoid checkouts

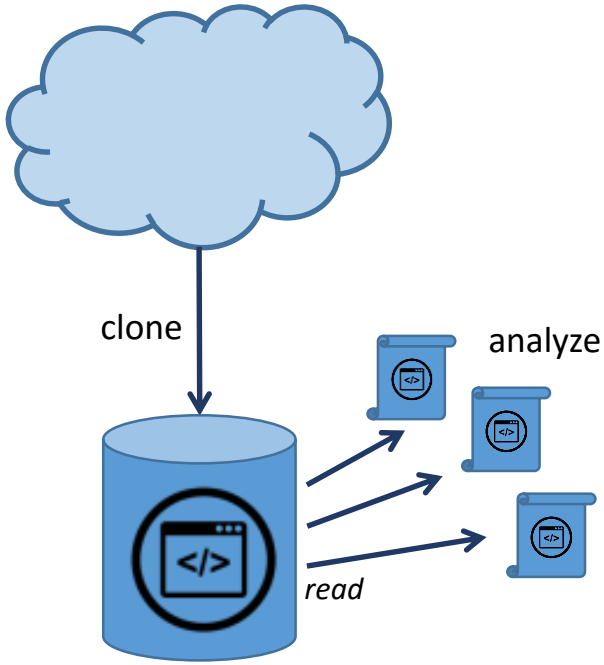


Avoid checkouts

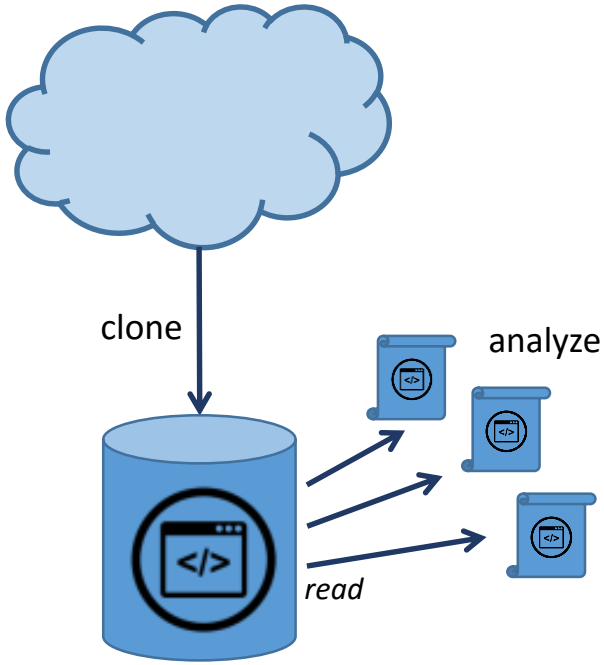


For every file: 2 read ops + 1 write op
Checkout includes irrelevant files
Need 1 CWD for every revision to be analyzed in parallel

Avoid checkouts

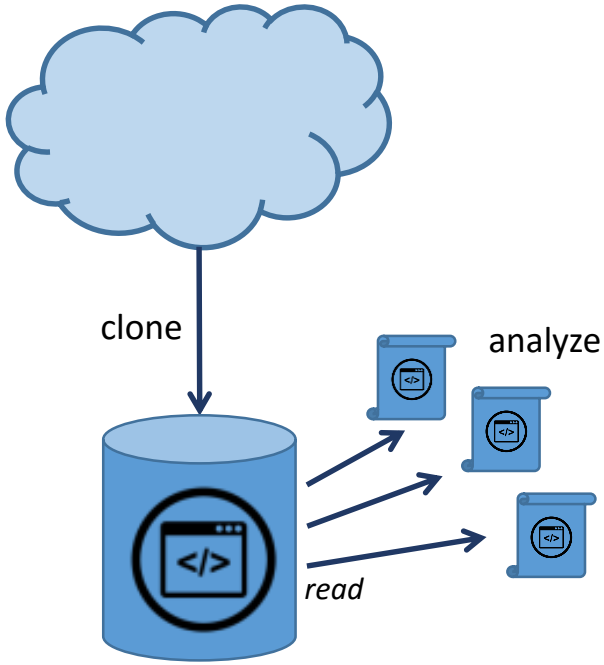


Avoid checkouts

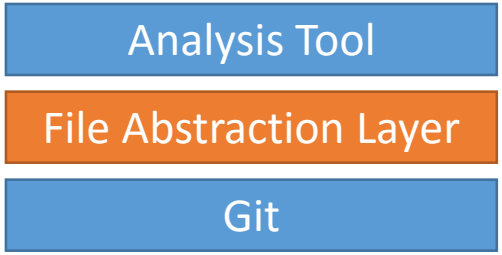


Only read relevant files in a single read op
No write ops
No overhead for parallization

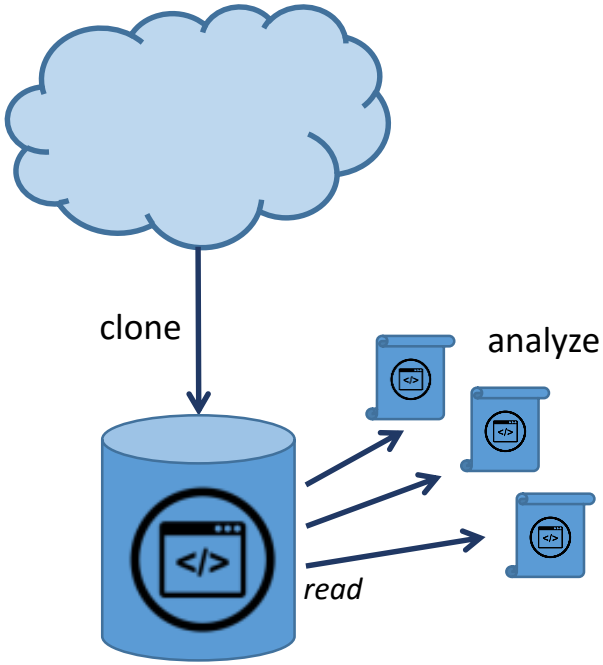
Avoid checkouts



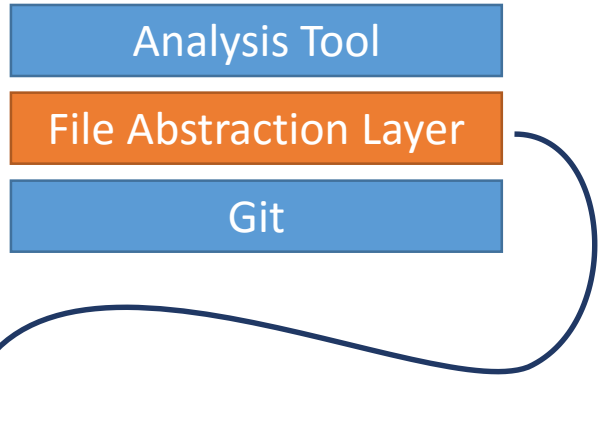
Only read relevant files in a single read op
No write ops
No overhead for parallization



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 JavaSourceFromCharArray(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 JavaSourceFromCharArray(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 

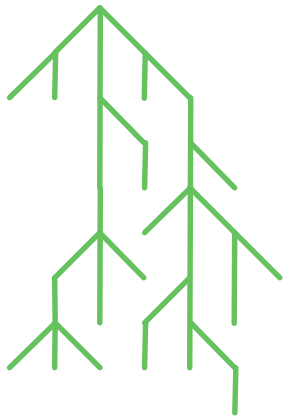
Merge ASTs

rev. 1

rev. 2

rev. 3

rev. 4



rev. 1

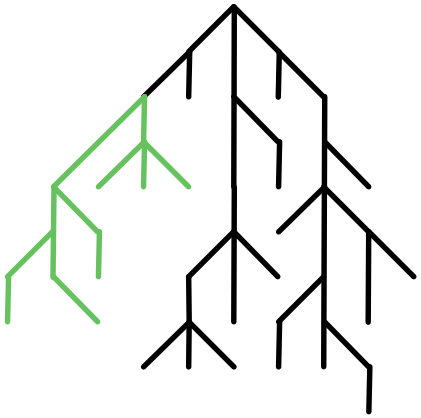
Merge ASTs

rev. 1

rev. 2

rev. 3

rev. 4



rev. 2

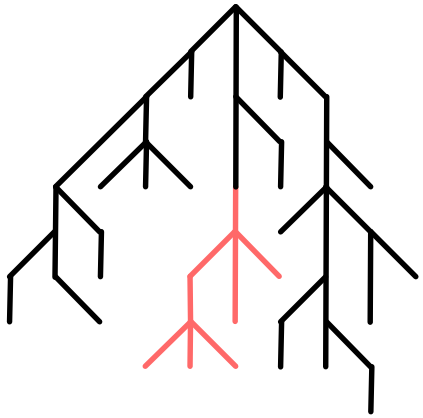
Merge ASTs

rev. 1

rev. 2

rev. 3

rev. 4



rev. 3

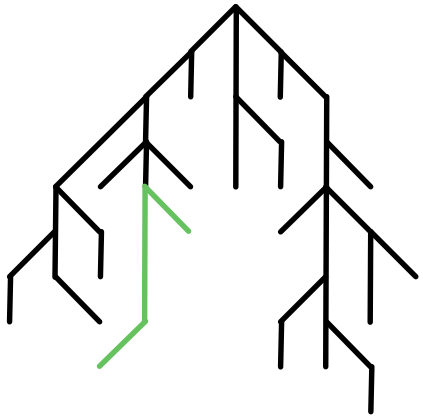
Merge ASTs

rev. 1

rev. 2

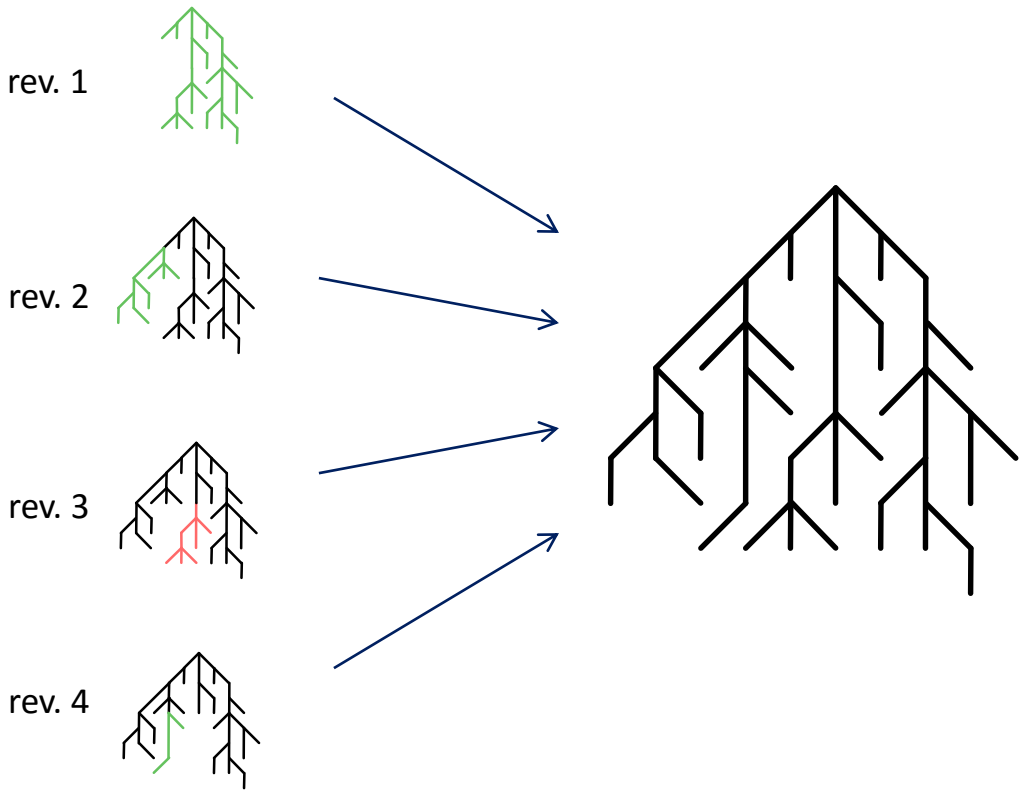
rev. 3

rev. 4

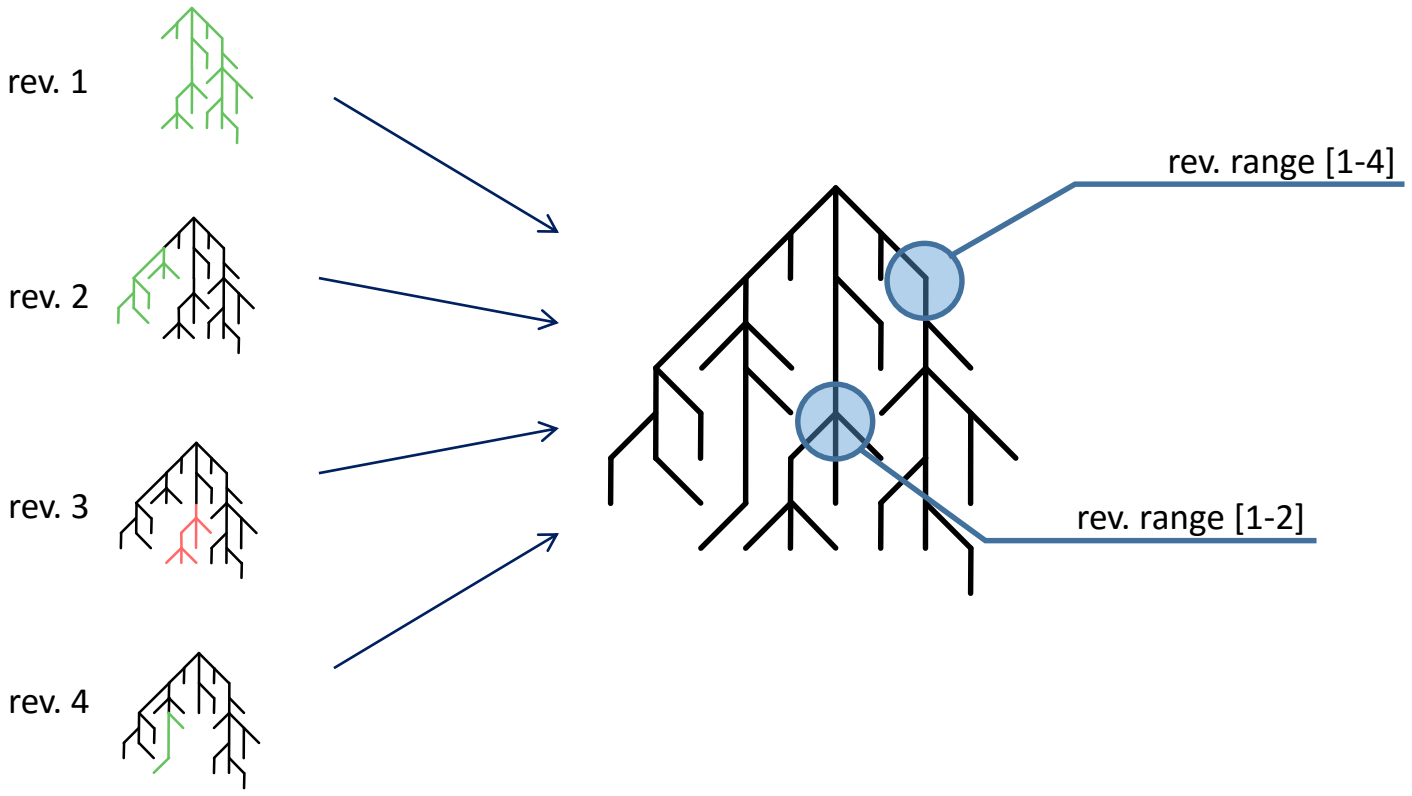


rev. 4

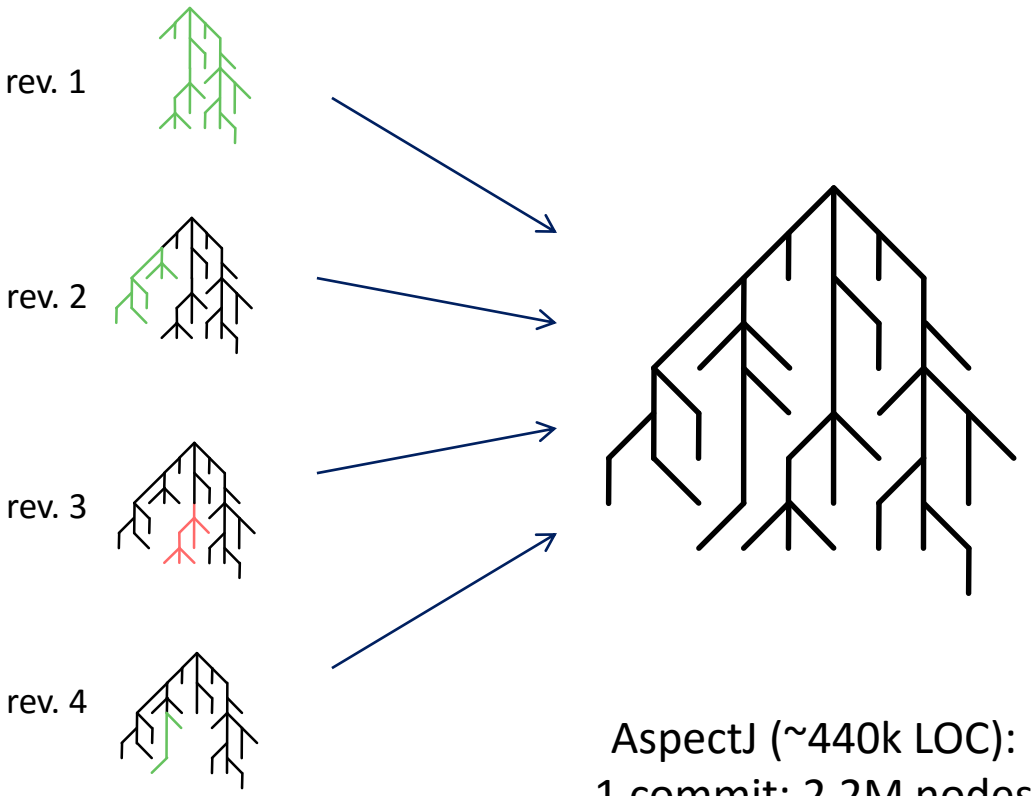
Merge ASTs



Merge ASTs



Merge ASTs



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

Merge ASTs

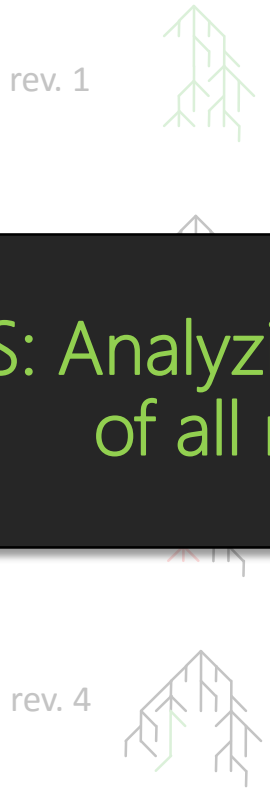
rev. 1

rev. 4

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)
      }
  }
}
```

parse

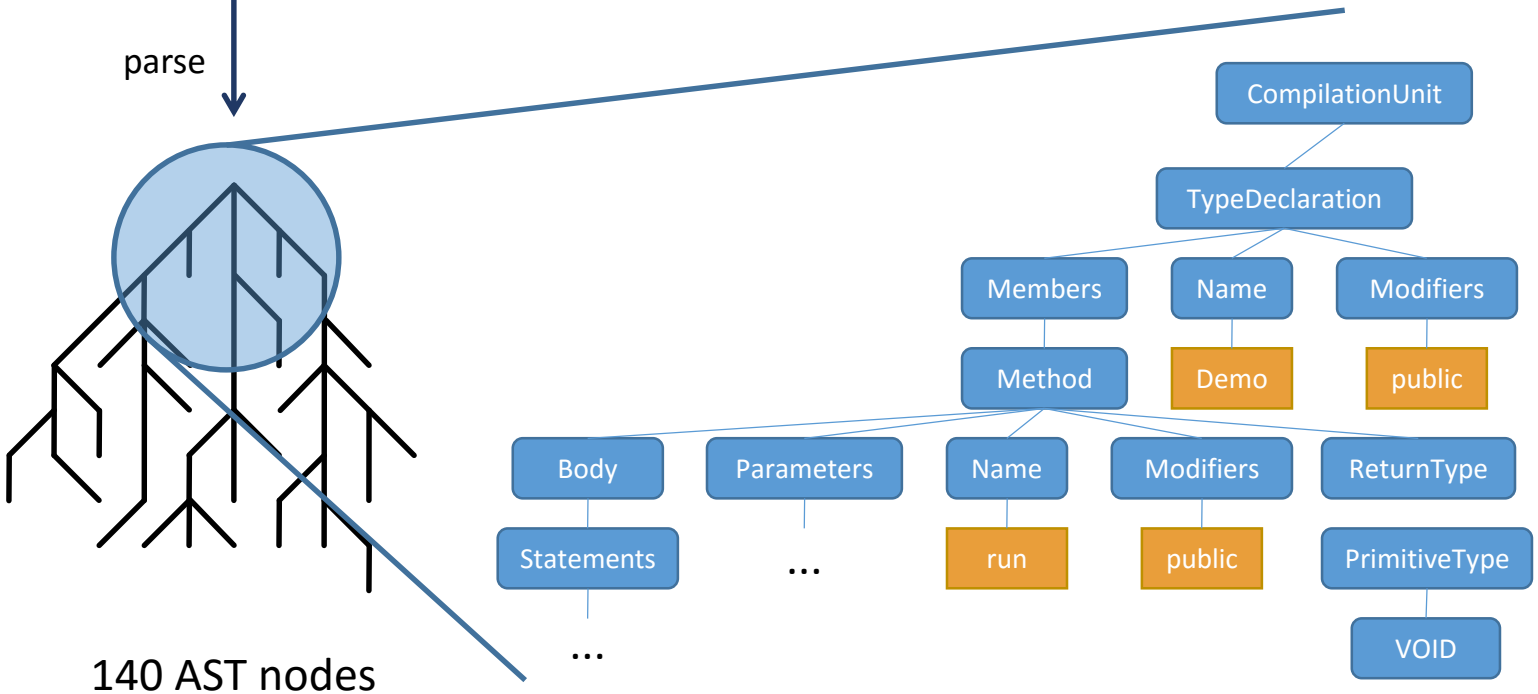
140 AST nodes
(using ANTLR)

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



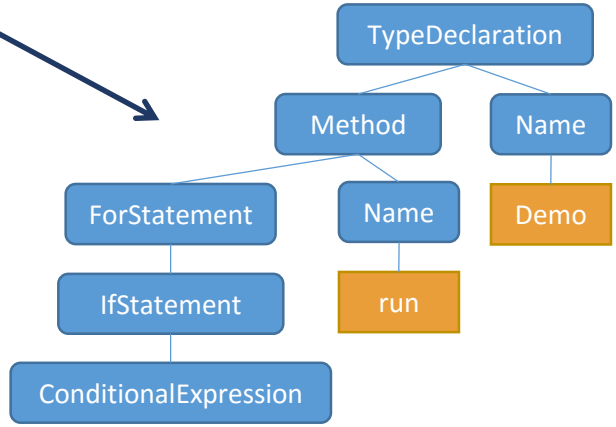
140 AST nodes
(using ANTLR)

```
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?

parse

filtered parse

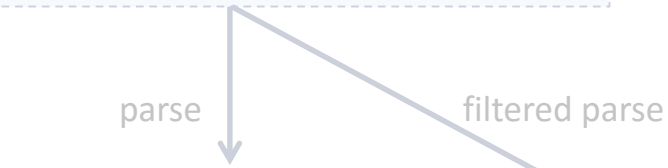


140 AST nodes
(using ANTLR)

7 AST nodes
(using ANTLR)

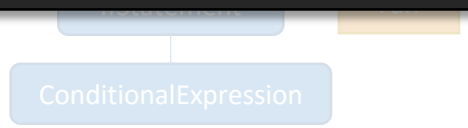
```
public class Demo {
  public void run() {
    for (int i = 1; i< 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
         System.out.println(i)
      }
  }
}
```

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



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

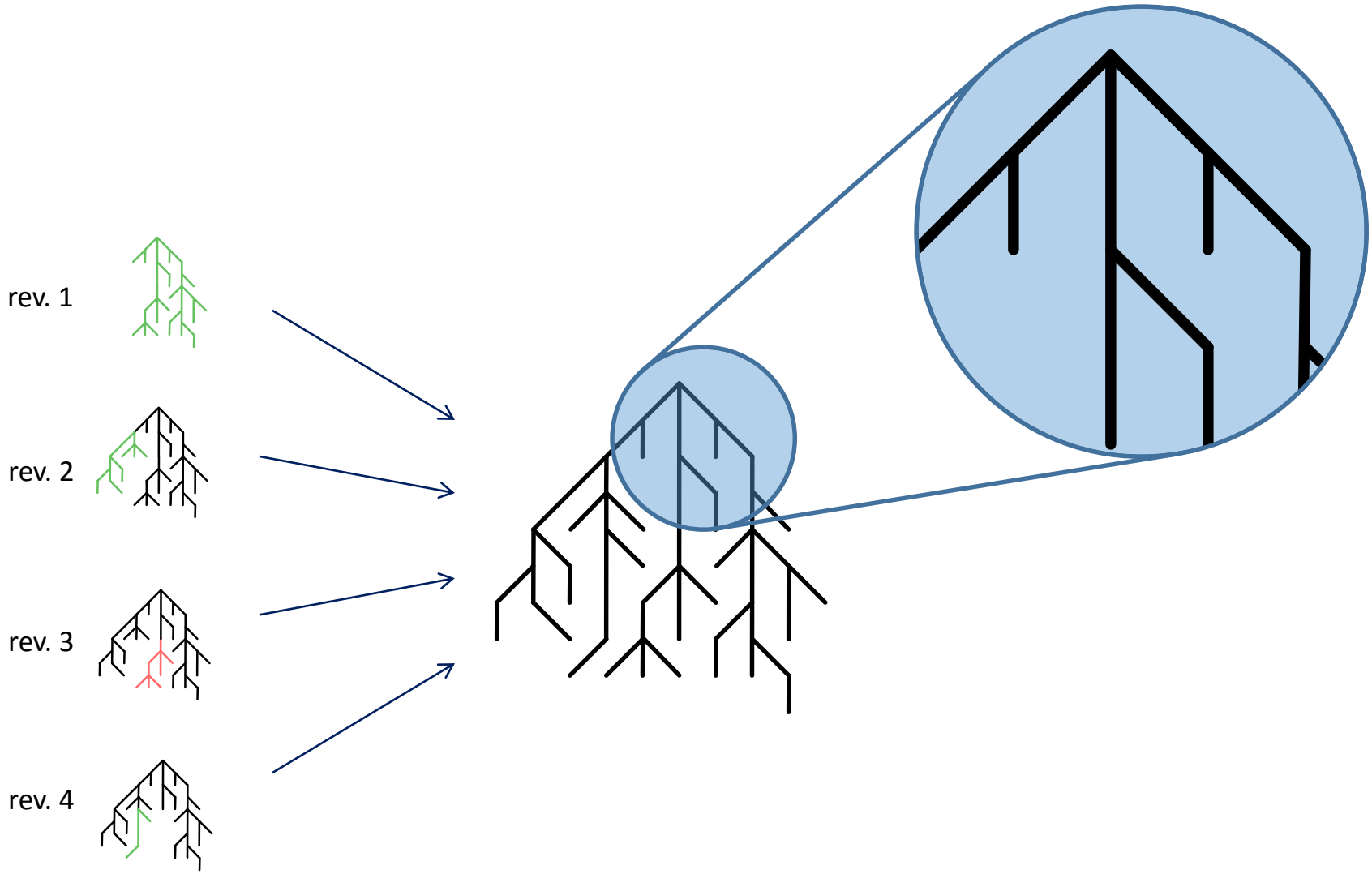
PS: Which AST nodes to load into the graph **depends on** the **analysis**

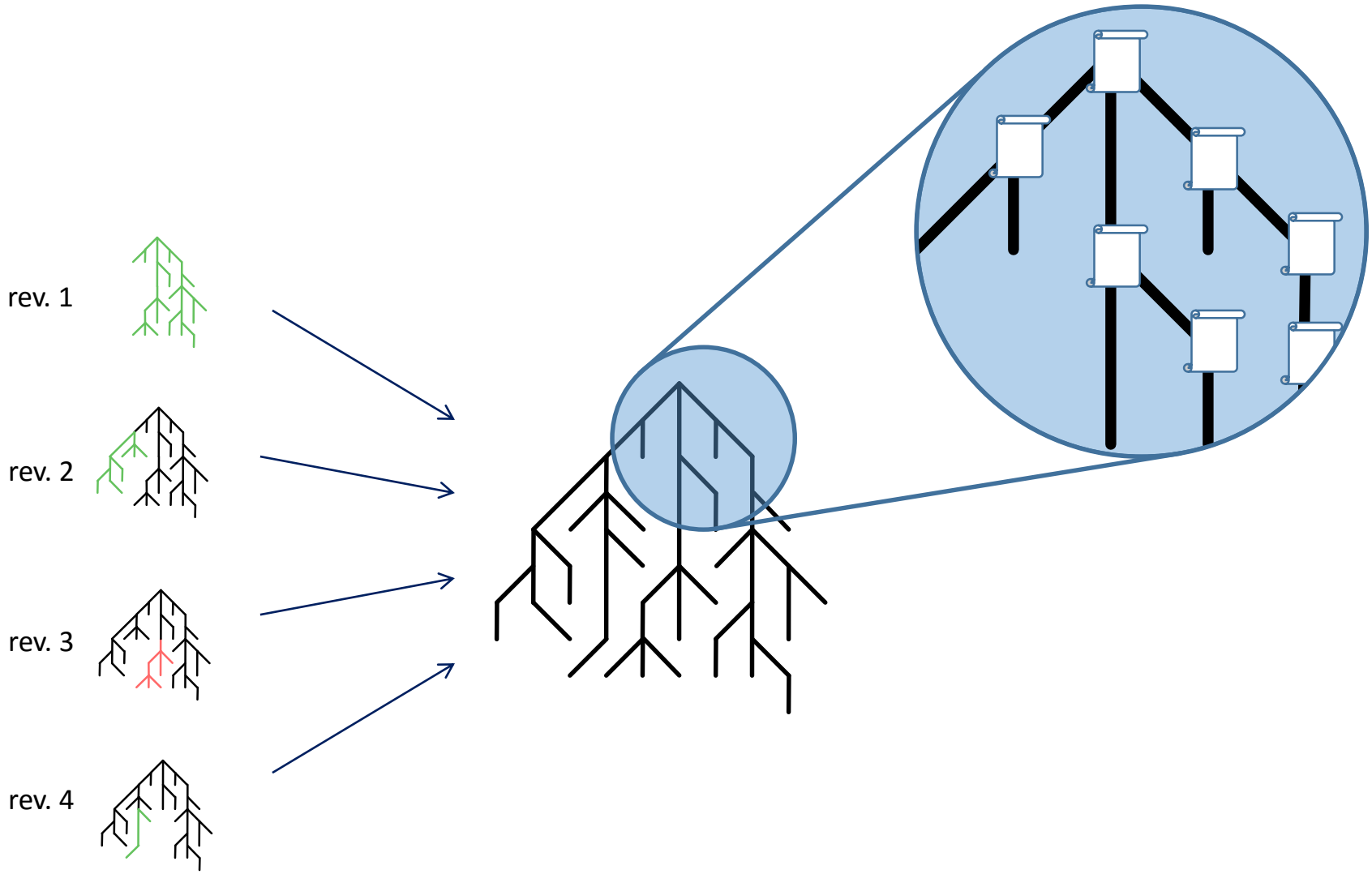
ConditionalExpression

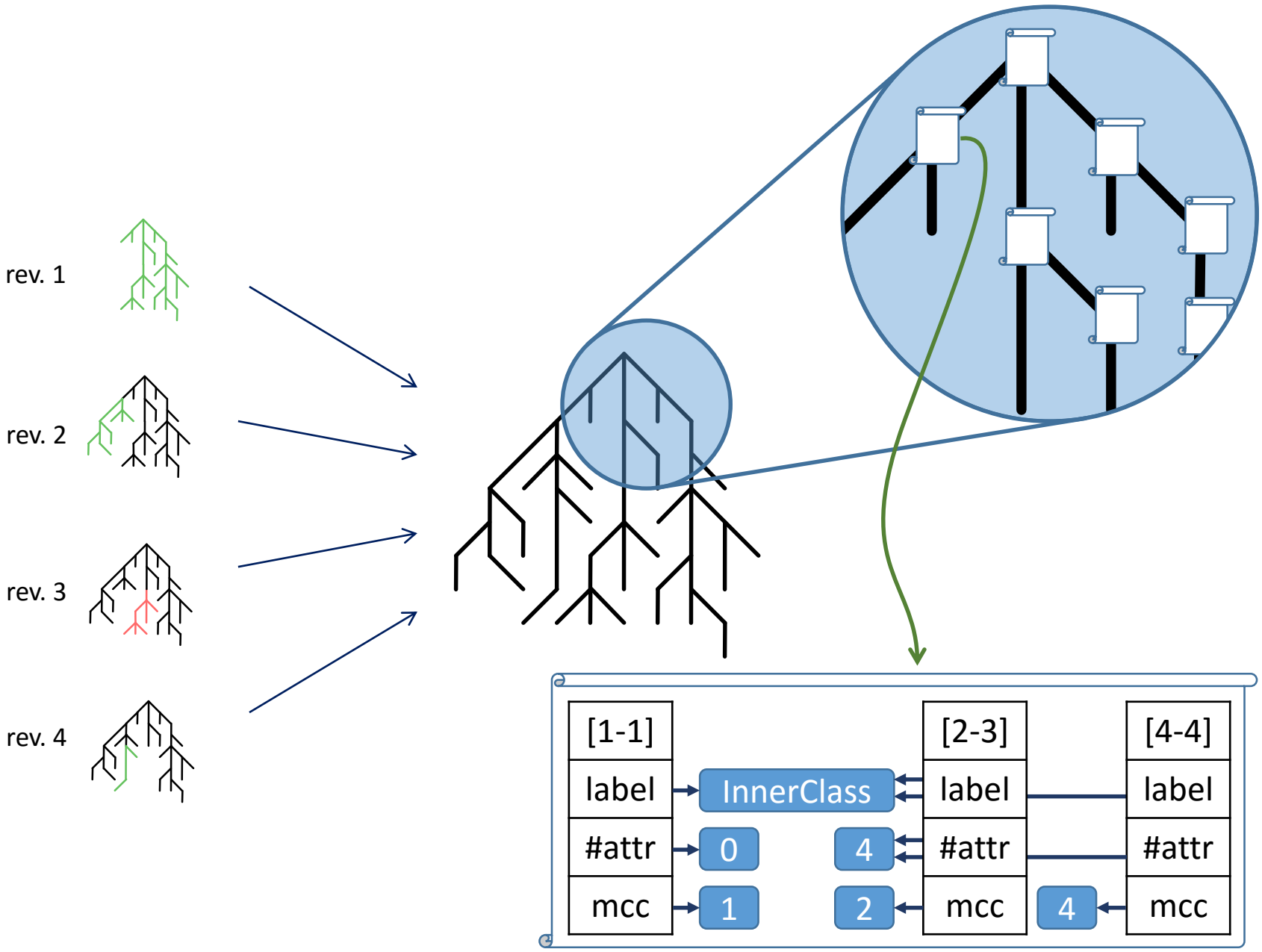
140 AST nodes
(using ANTLR)

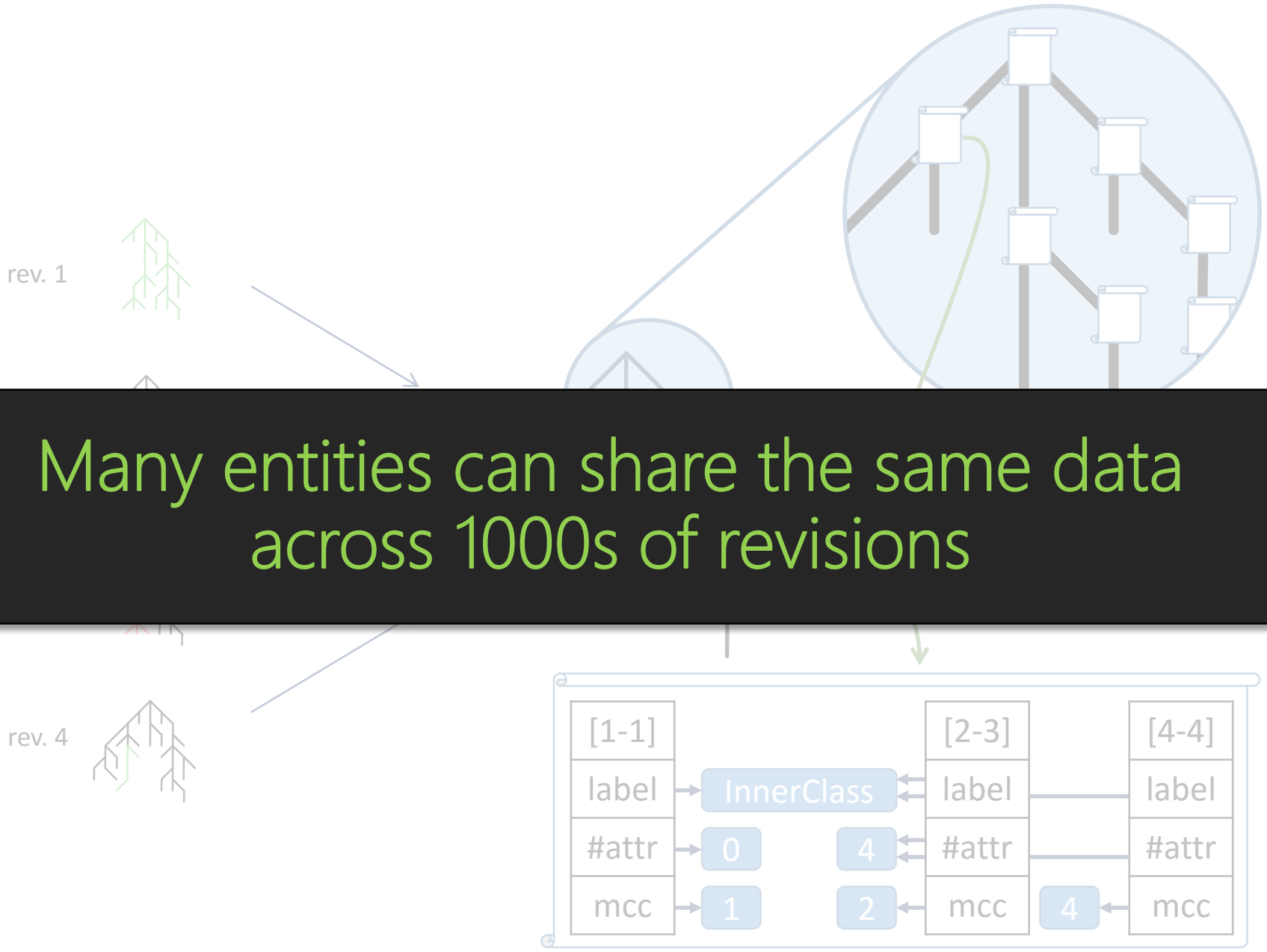
7 AST nodes
(using ANTLR)

#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
[2-3]
label
#attr
mcc
4
2
[4-4]
label
#attr
mcc
4



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

LISA also does:

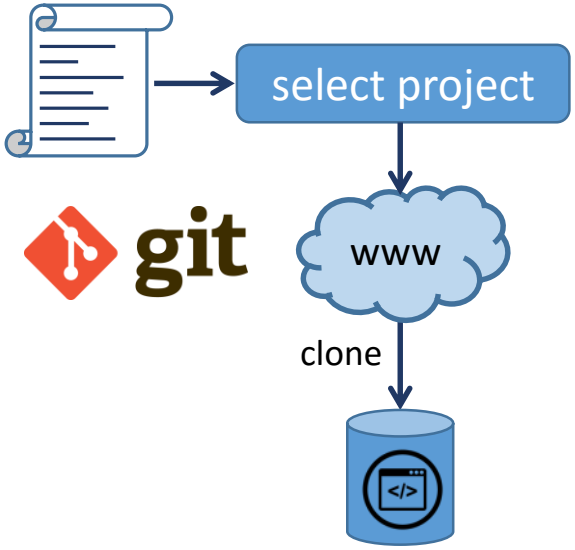
#5: Parallel Parsing

#6: Asynchronous graph computation

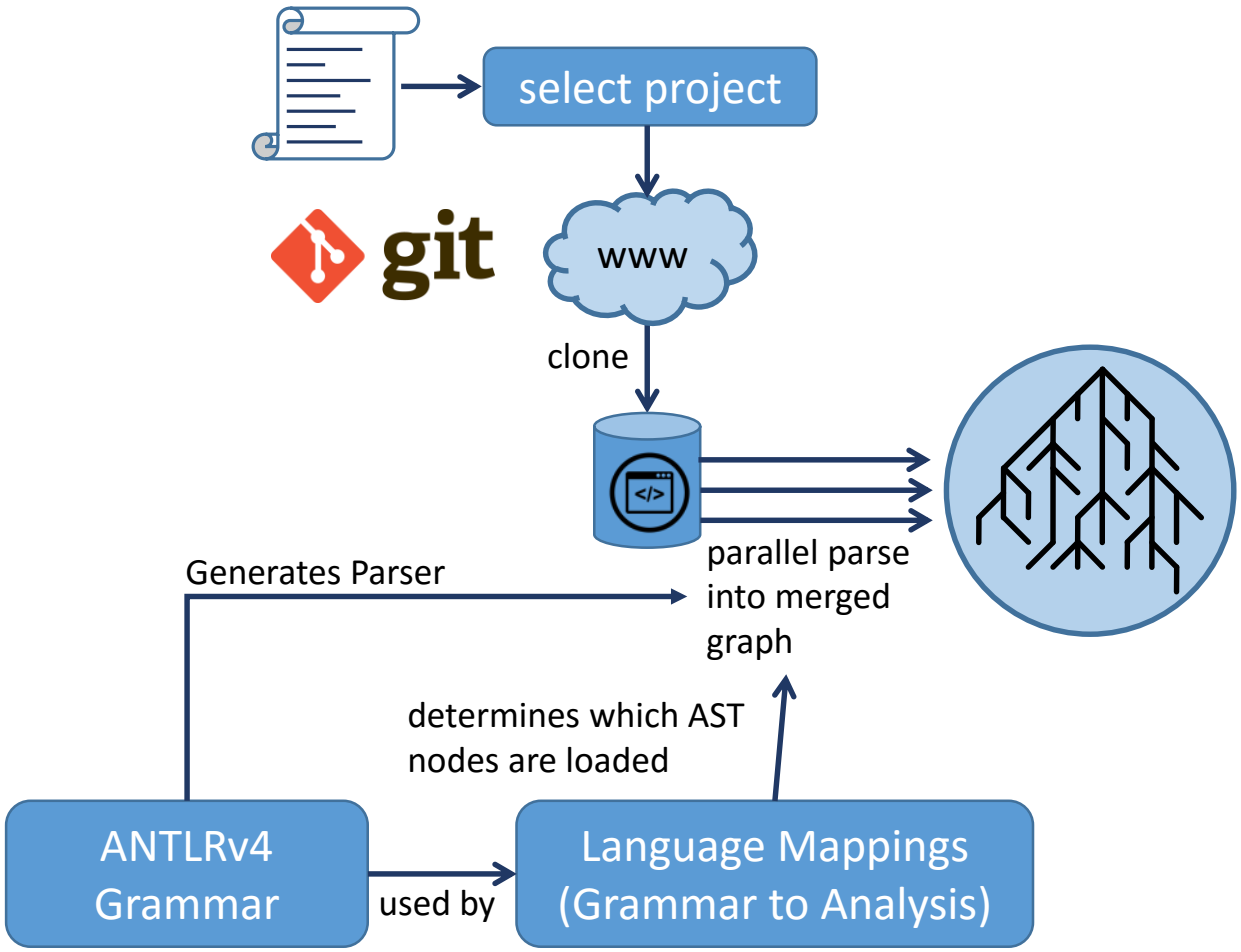
#7: **Generic graph computations** applying to ASTs from **compatible languages**

To Summarize...

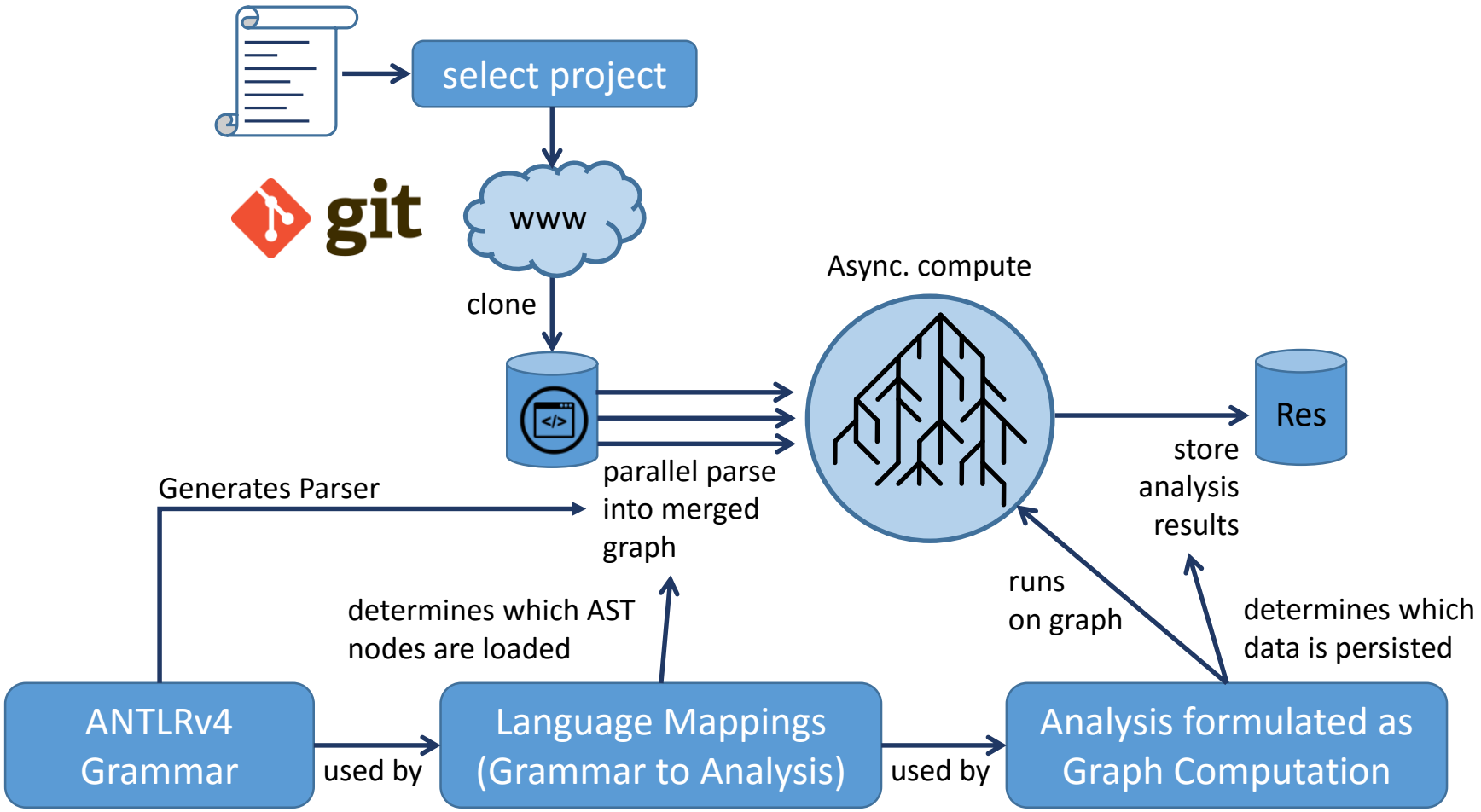
The LISA Analysis Process



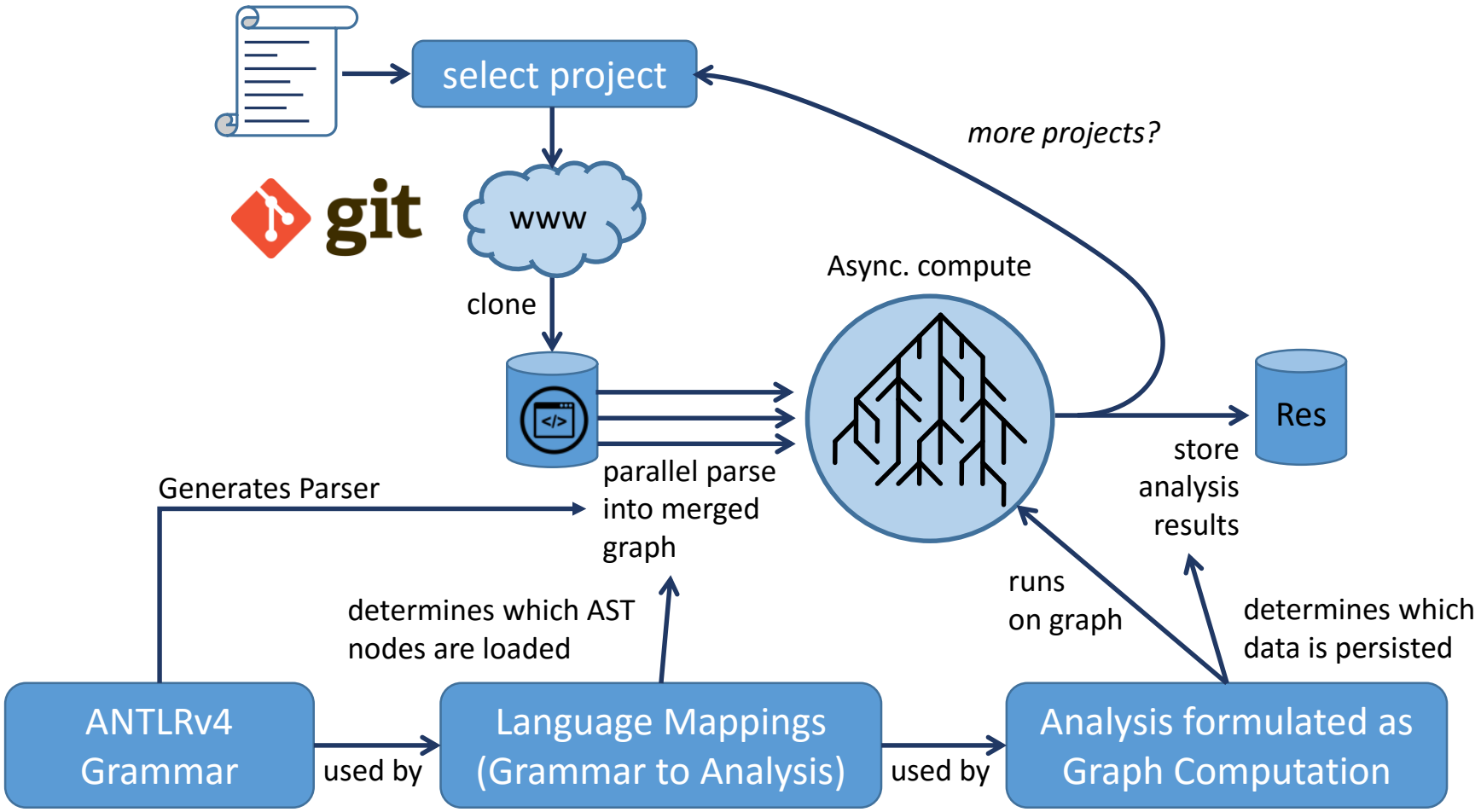
The LISA Analysis Process



The LISA Analysis Process

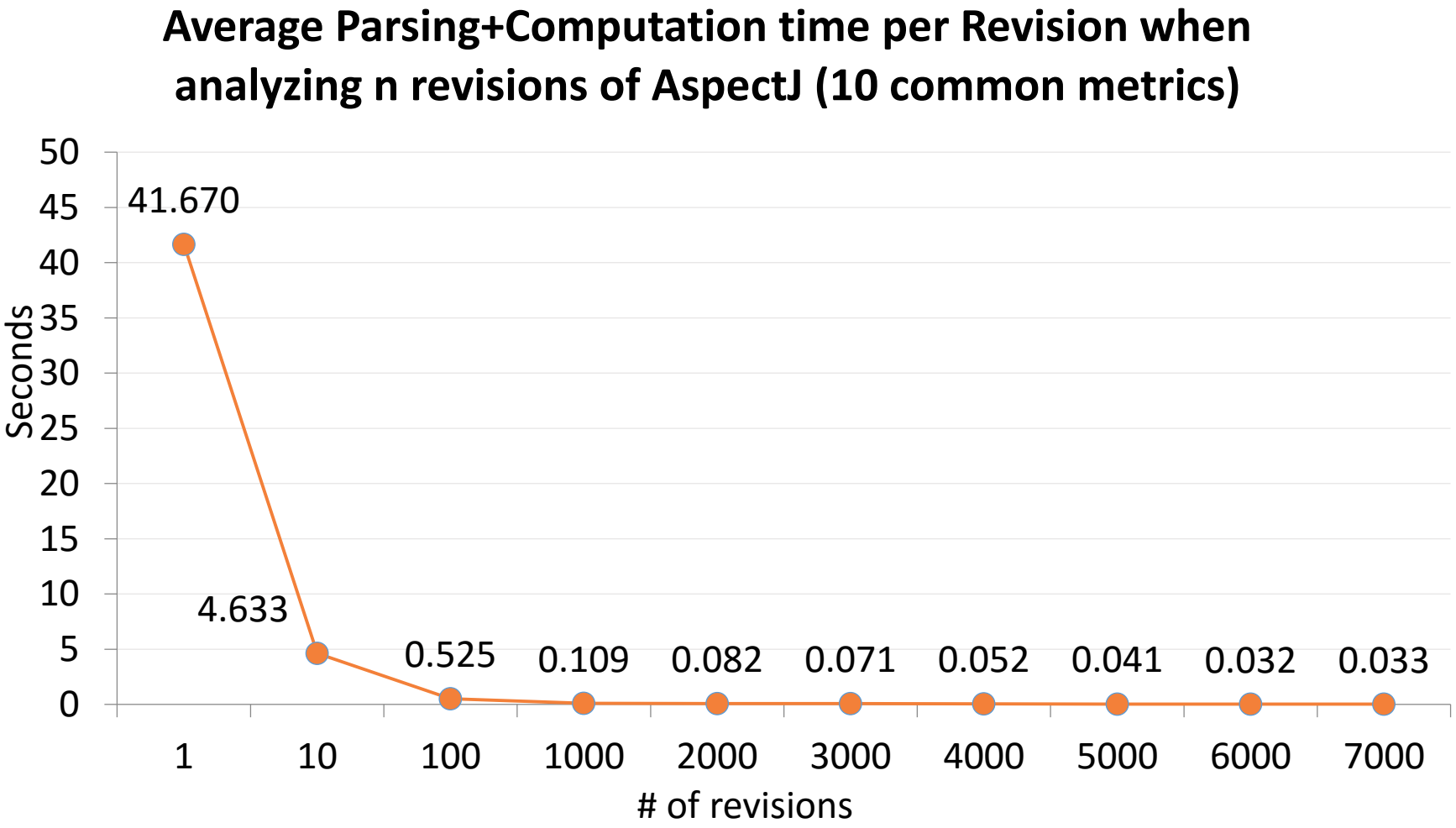


The LISA Analysis Process



How well does it work, then?

Marginal cost for +1 revision



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...)

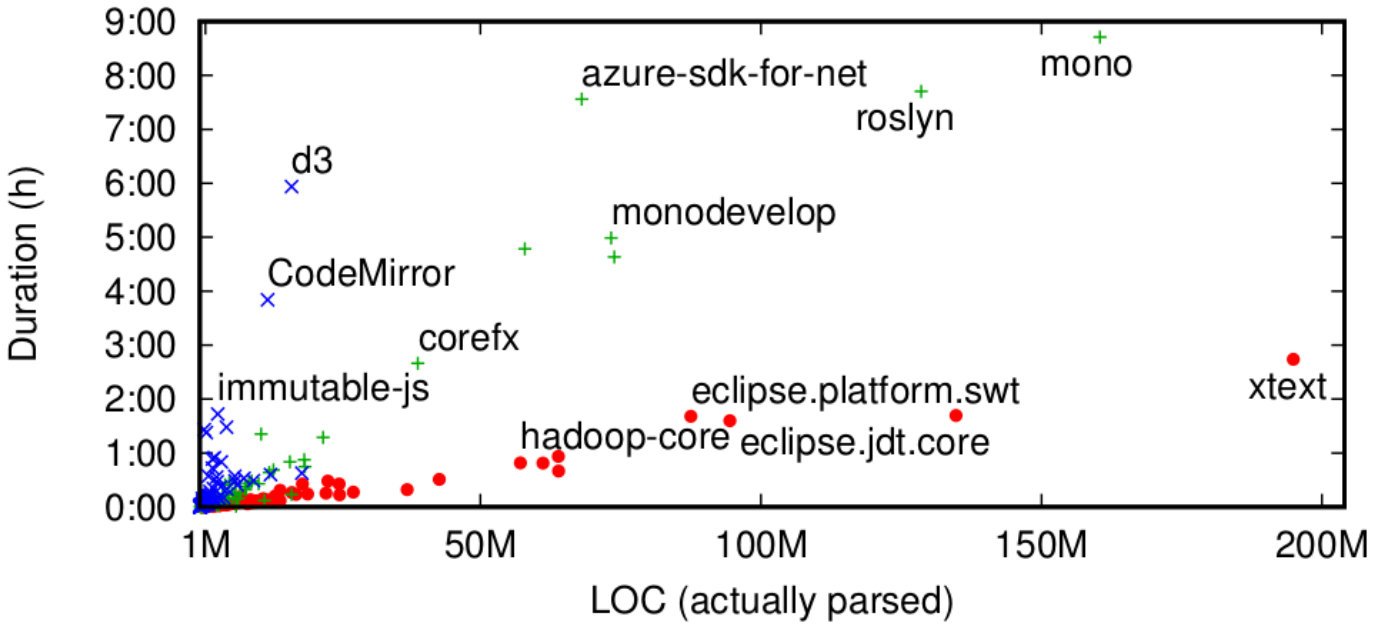
The (not so) minor stuff

- 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)

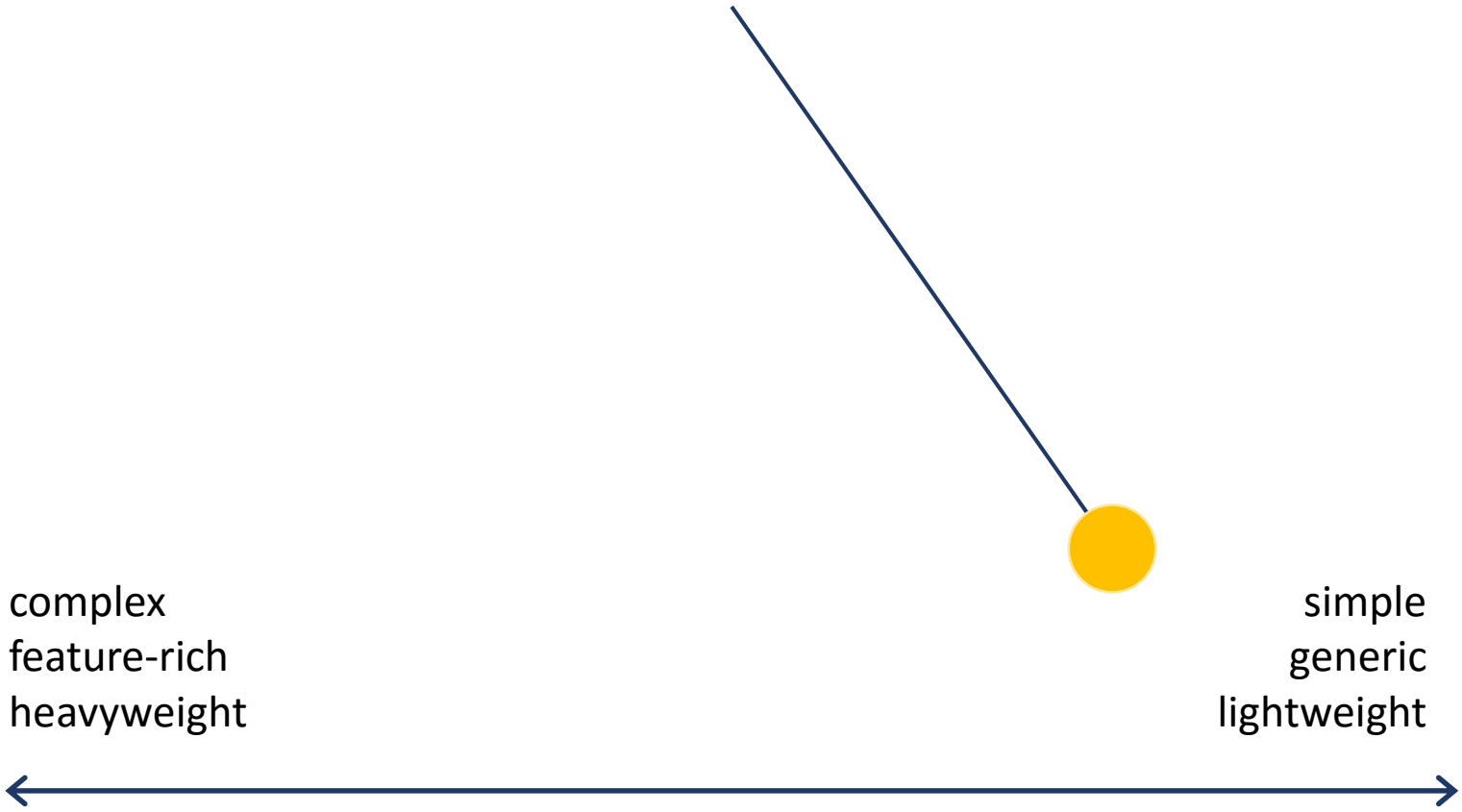
Language matters



E.g.: Javascript takes longer because:

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

LISA is EXTREME



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