University of Zurich[UZH]

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

Guest Talk
@ MIT CSAIL
Boston, Massachusetts, USA

Light-weight static analysis of multi-language, multi-revision artifacts

Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, Harald C. Gall
Software Evolution and Architecture Lab
University of Zurich, Switzerland
{alexandru,panichella,proksch,gall}@ifi.uzh.ch
05.11.2018

Carol V. Alexandru

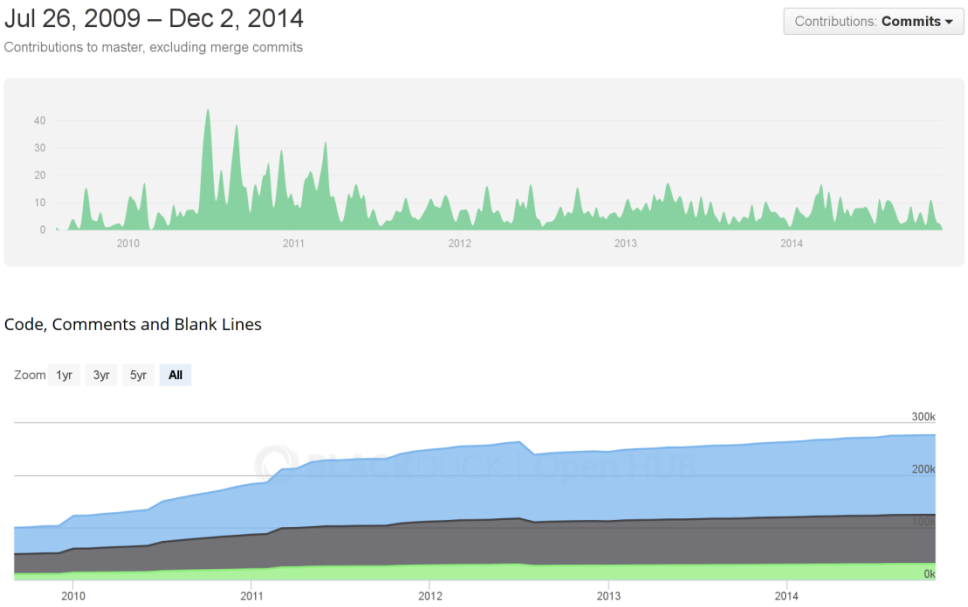
Sebastiano Panichella

Sebastian Proksch

Harald C. Gall

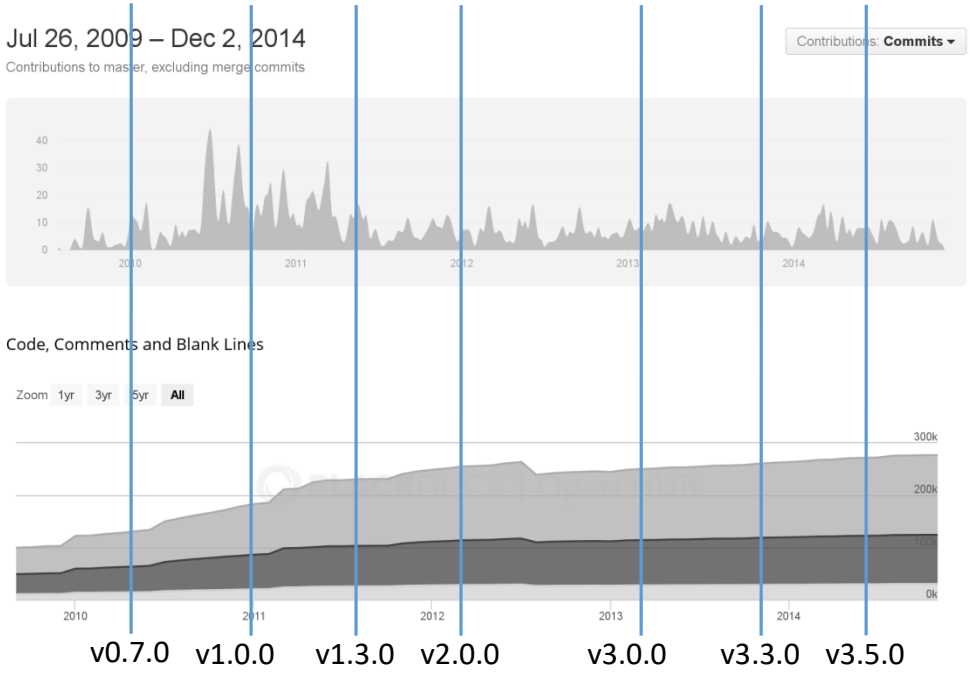
The Problem Domain

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



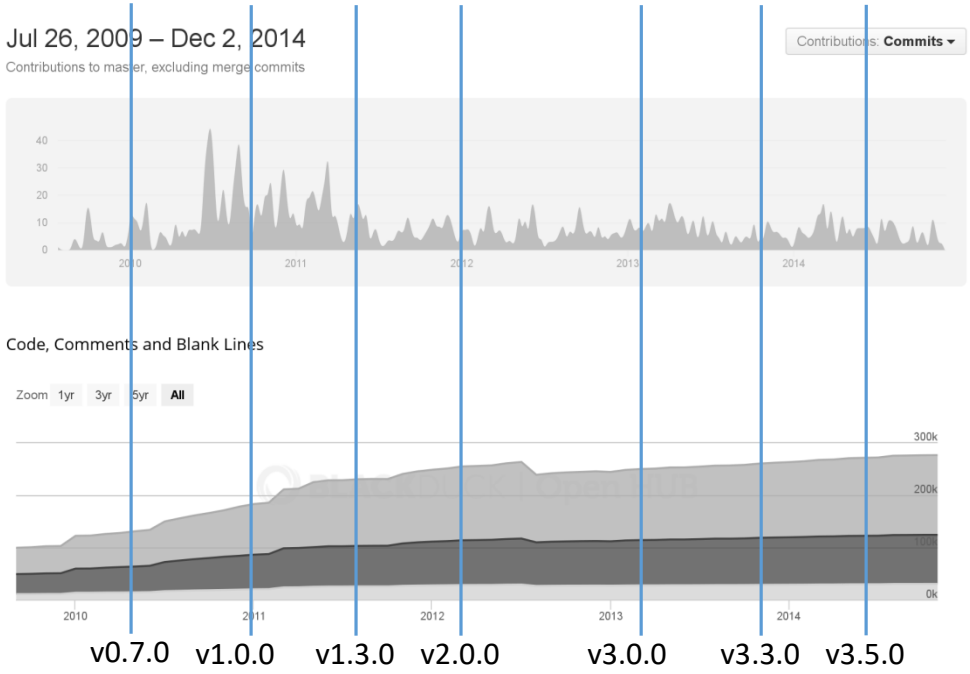
The Problem Domain

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

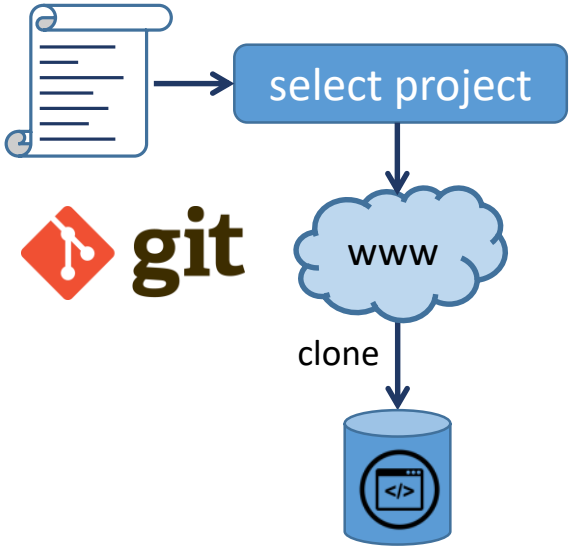


The Problem Domain

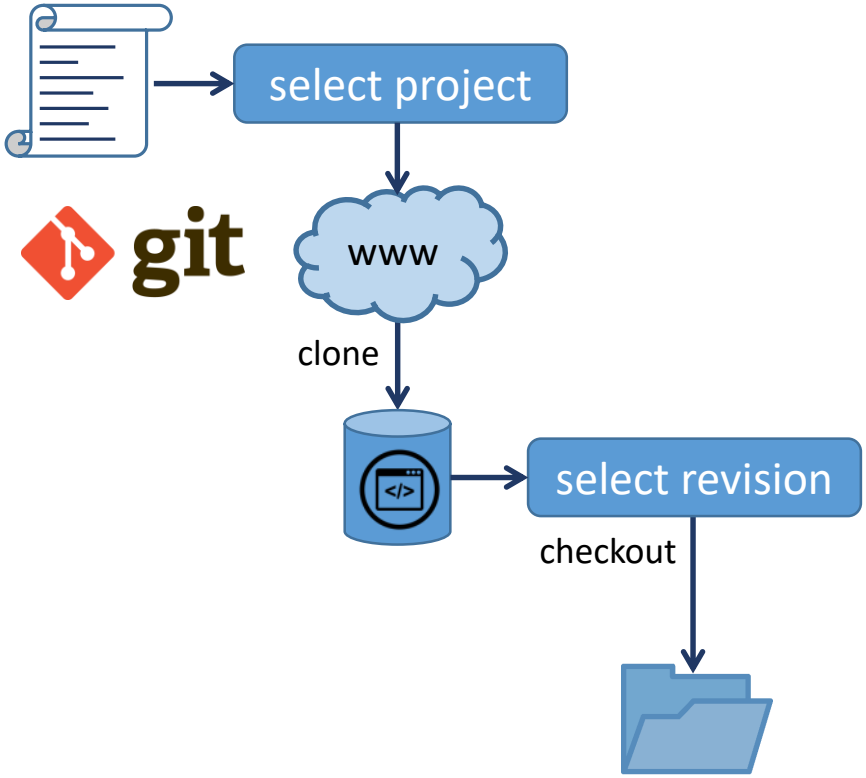
- Static analysis (e.g. #Attr., McCabe, coupling...)
- Many revisions, fine-grained historical data



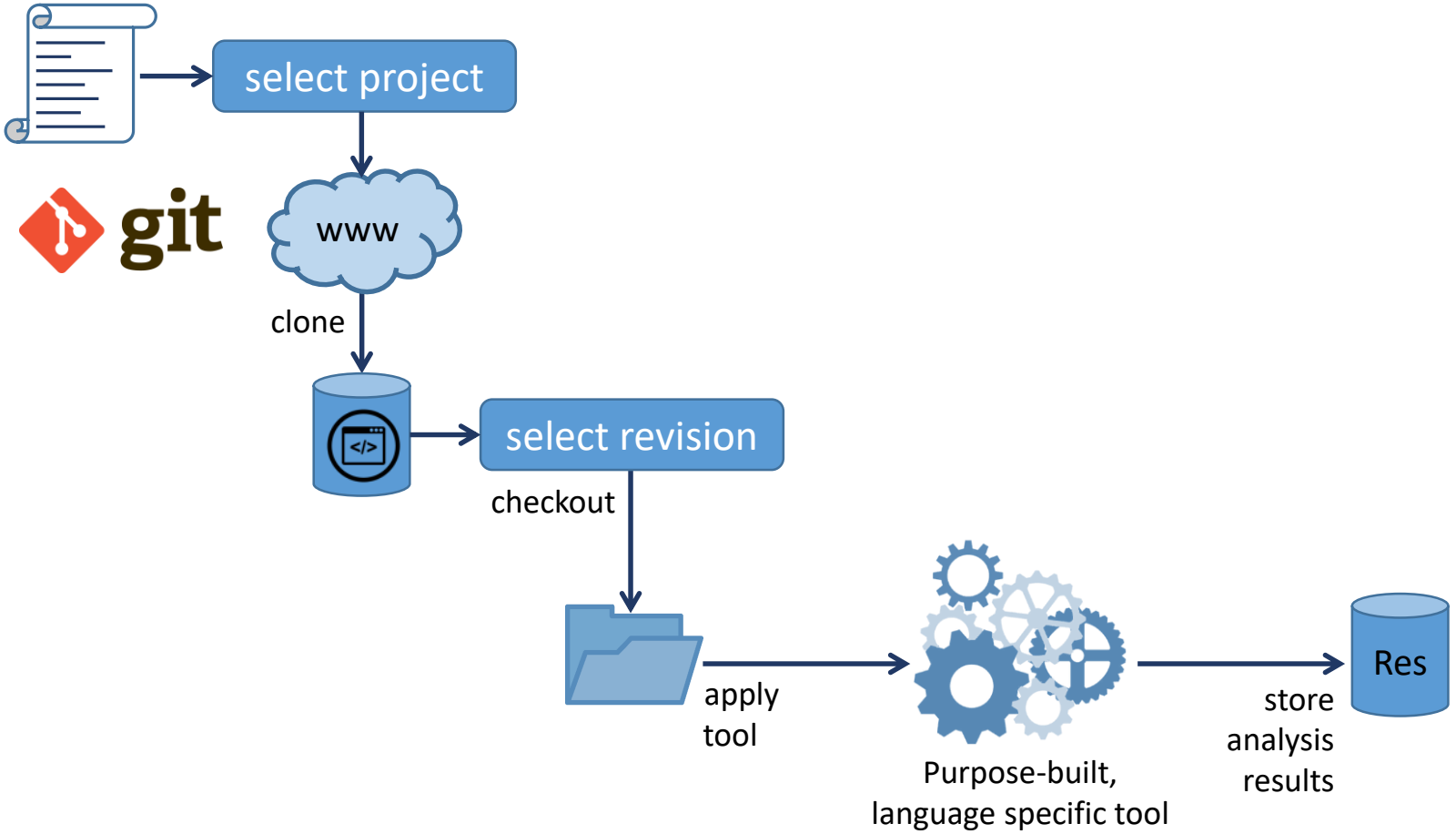
A Typical Analysis Process



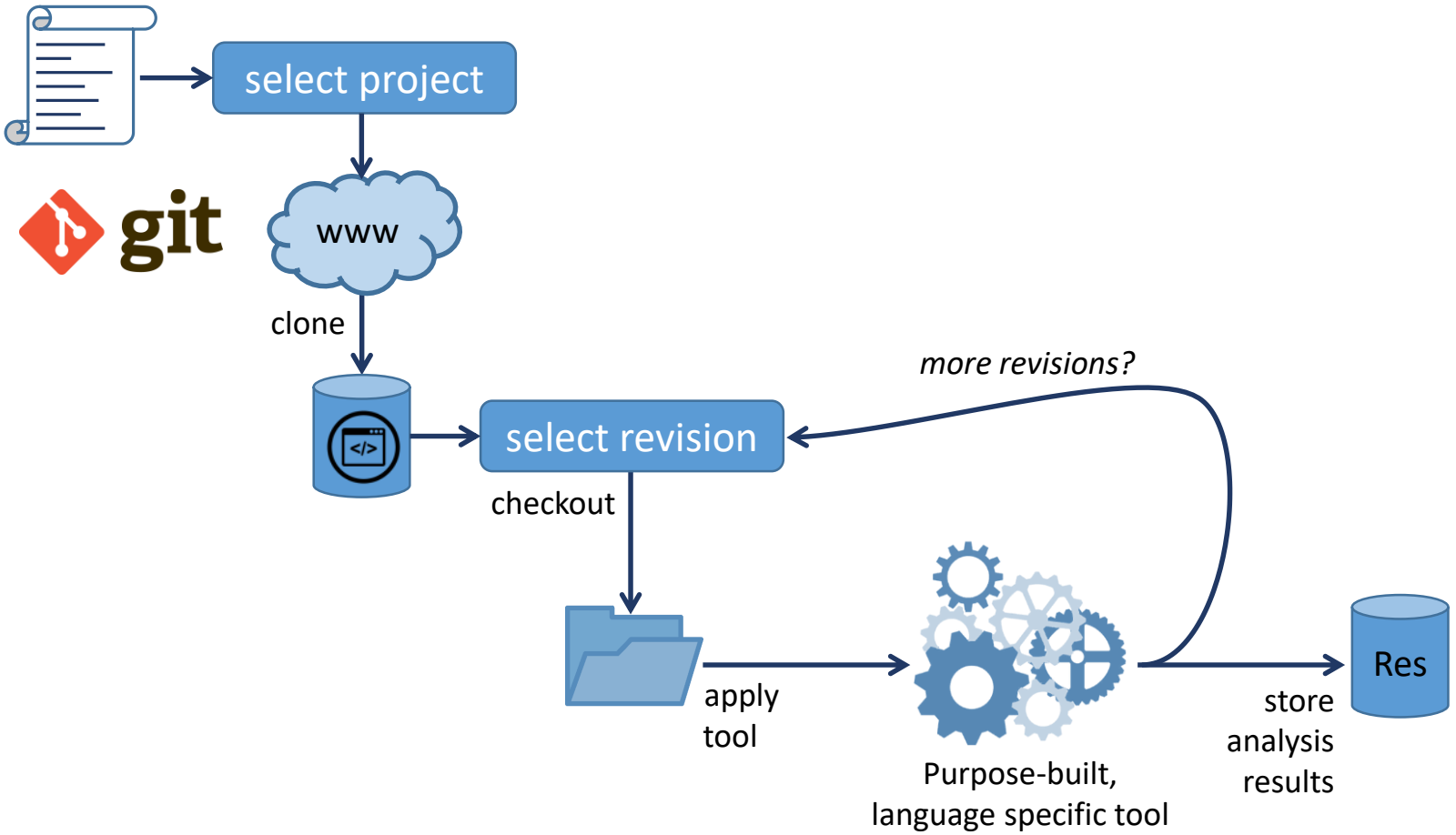
A Typical Analysis Process



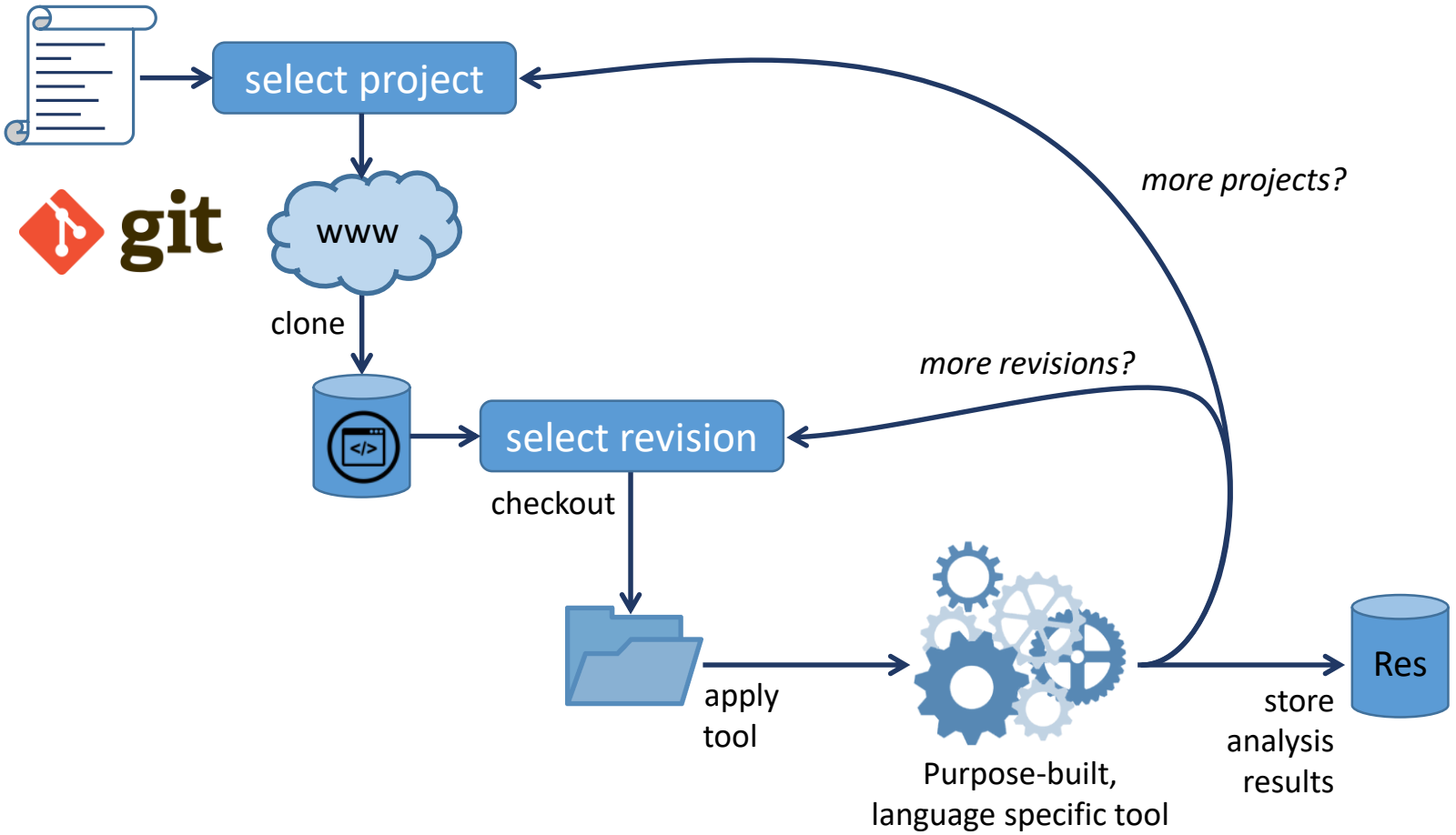
A Typical Analysis Process



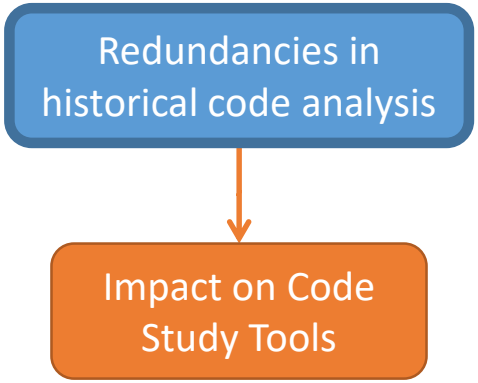
A Typical Analysis Process



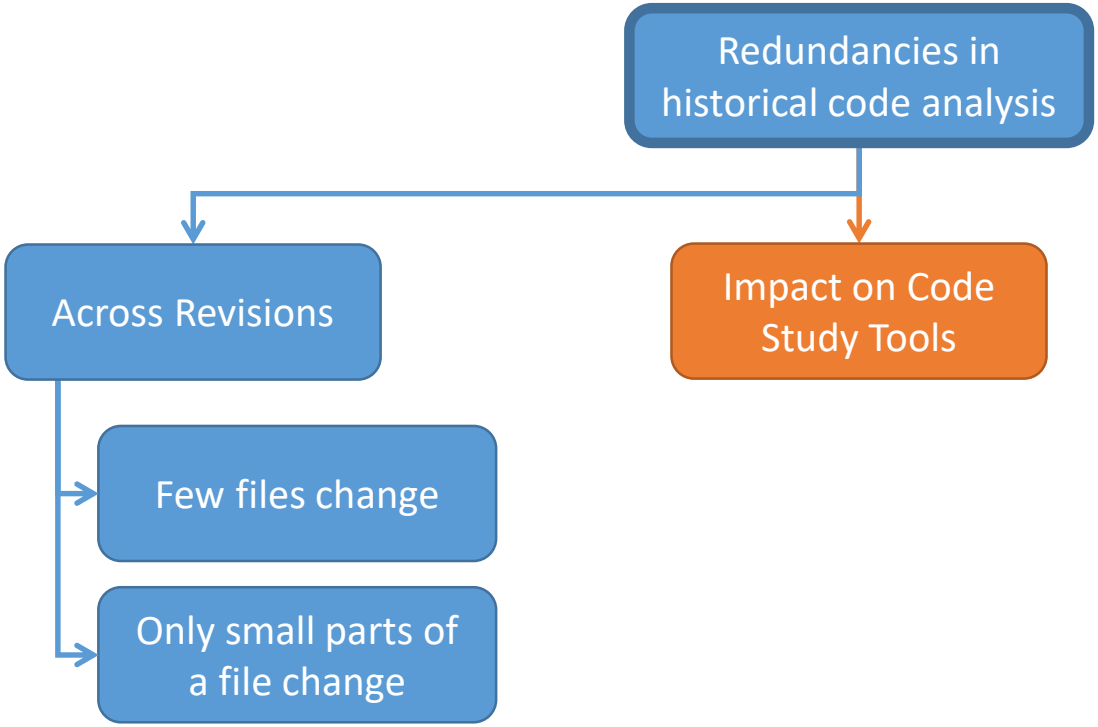
A Typical Analysis Process



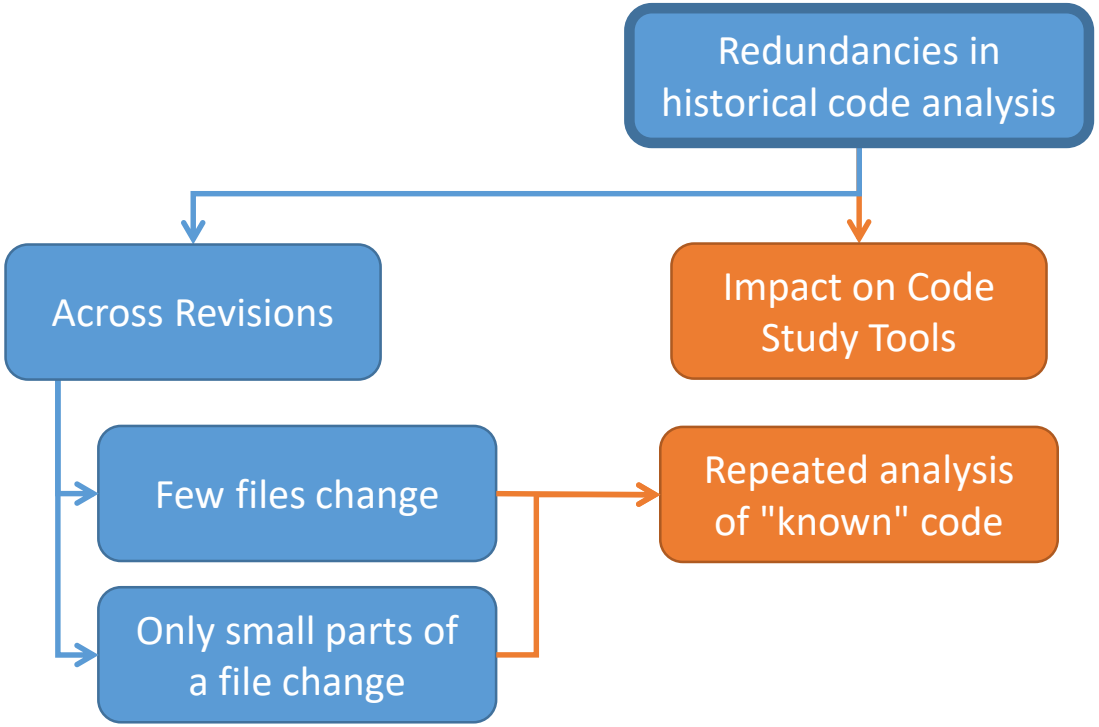
Redundancies all over...



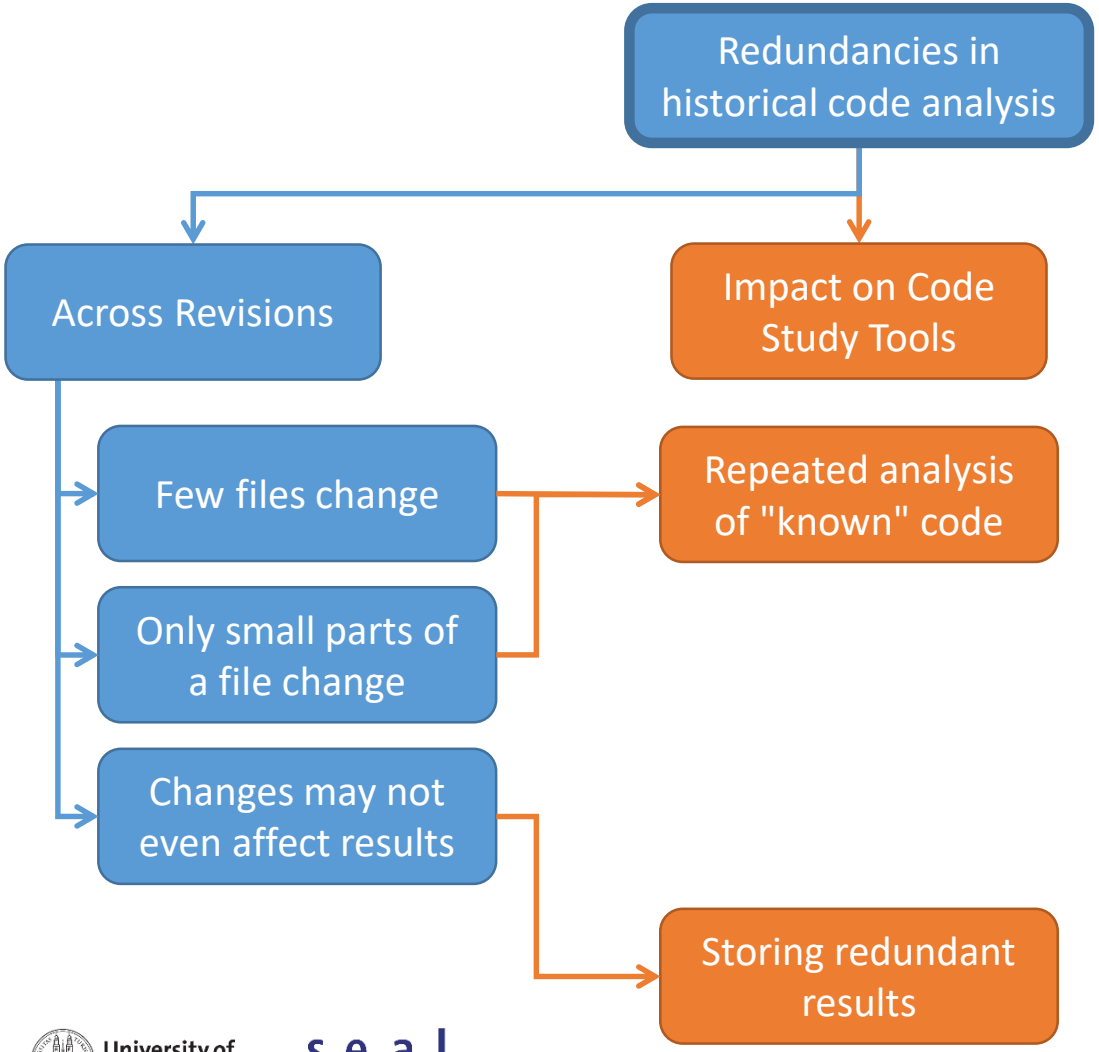
Redundancies all over...



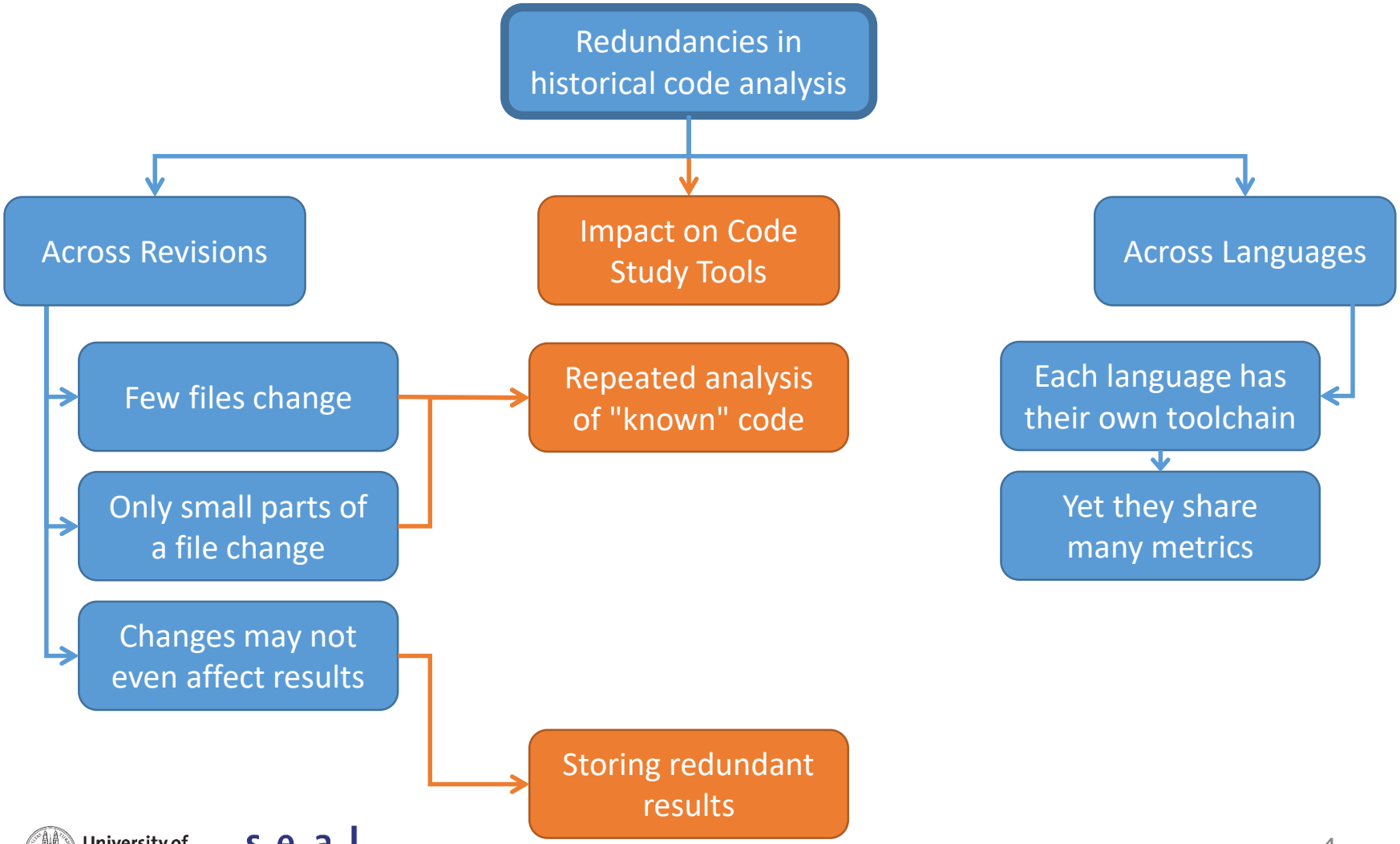
Redundancies all over...



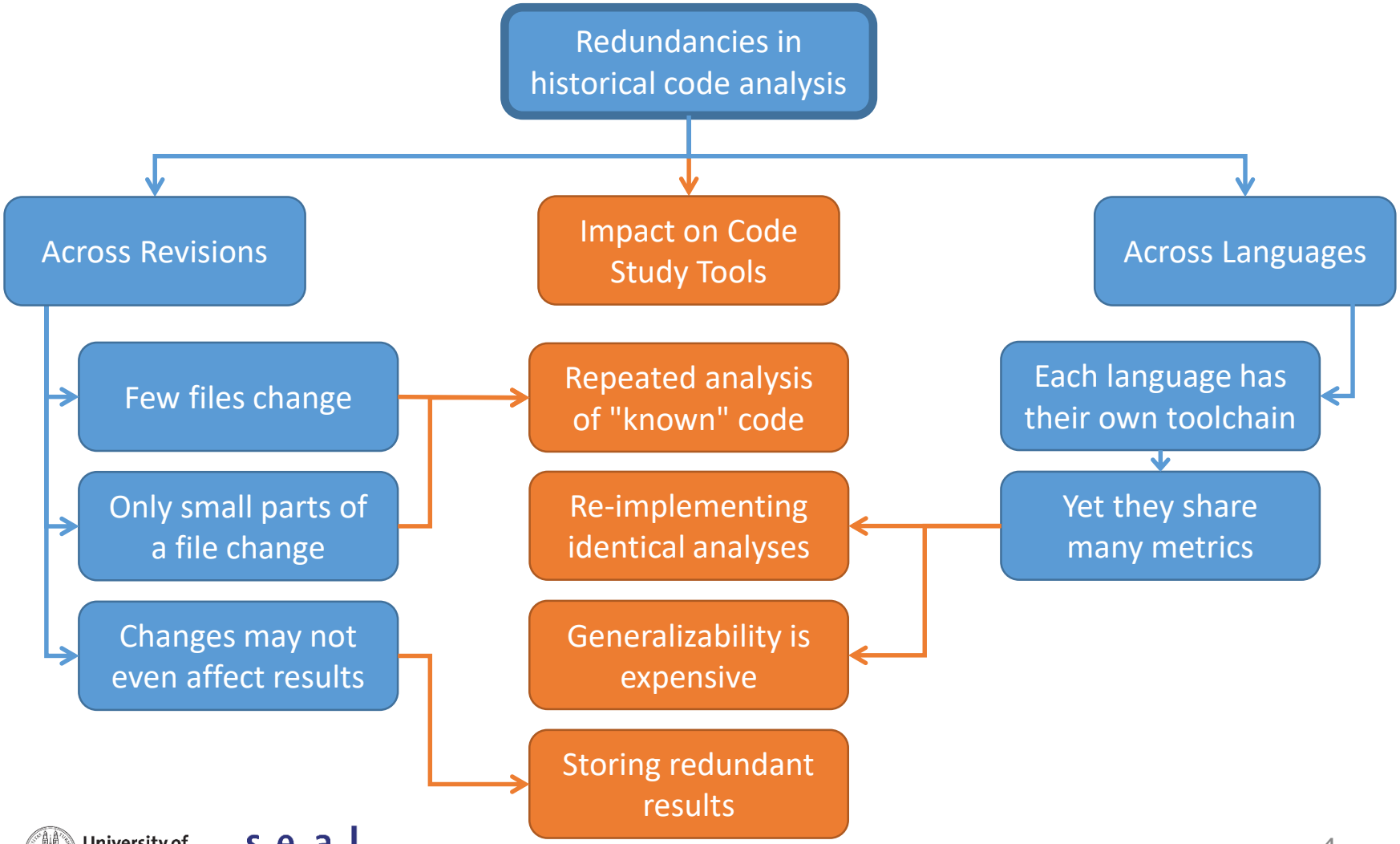
Redundancies all over...



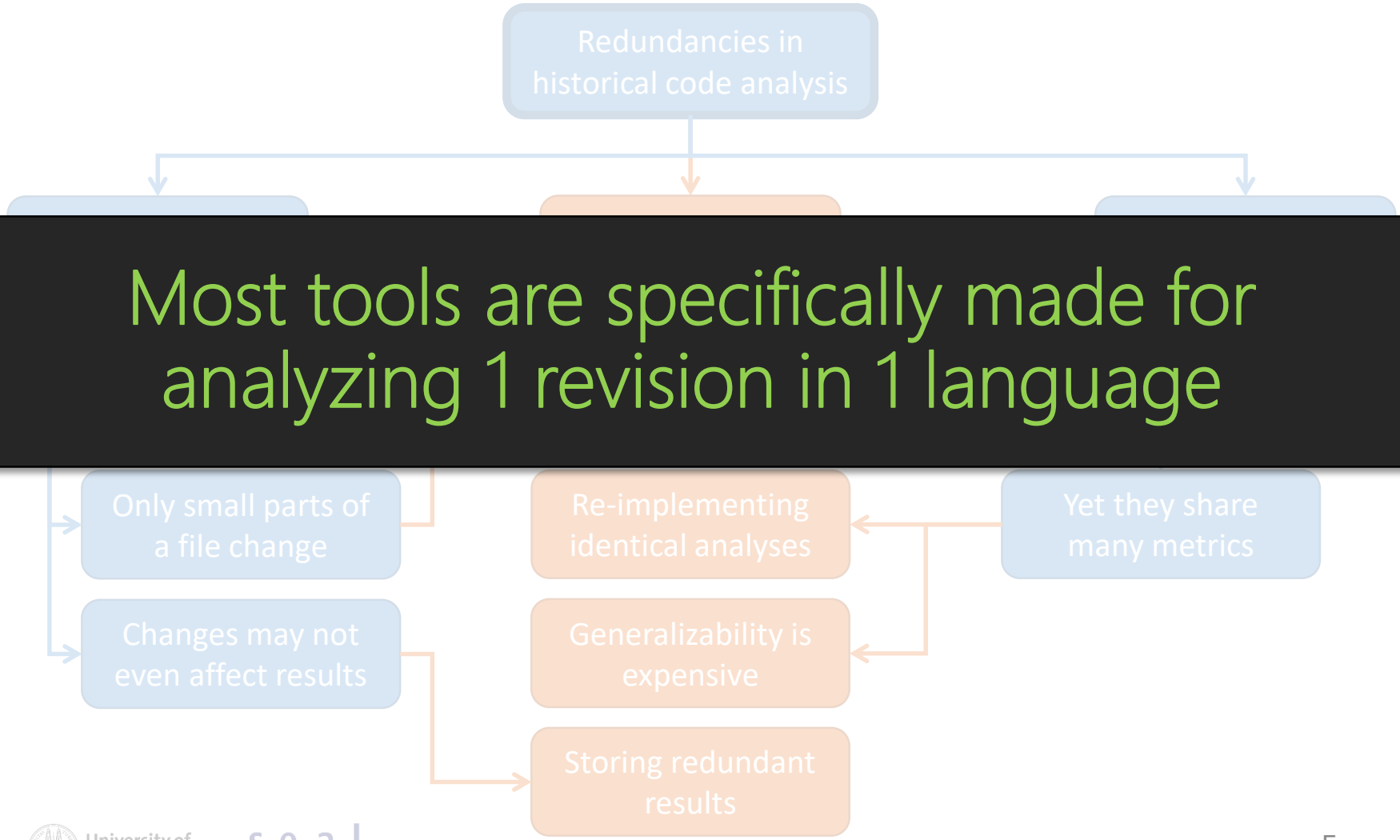
Redundancies all over...



Redundancies all over...

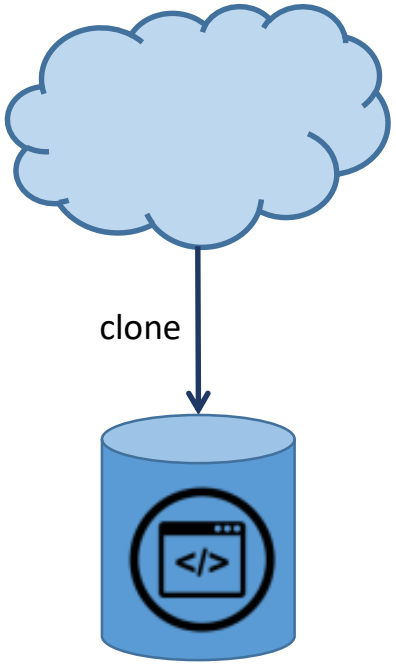


Redundancies all over…

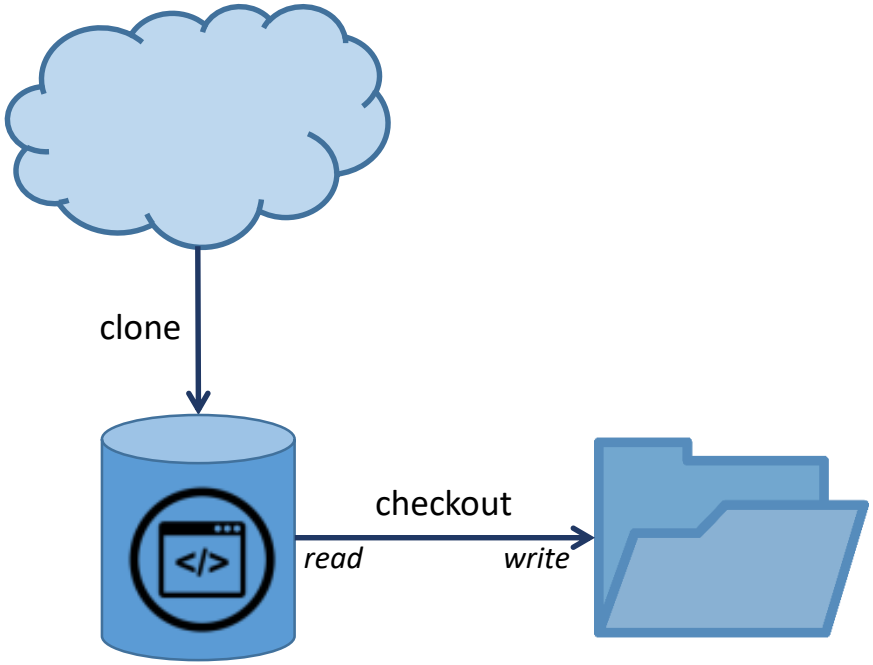


#1: Avoid Checkouts

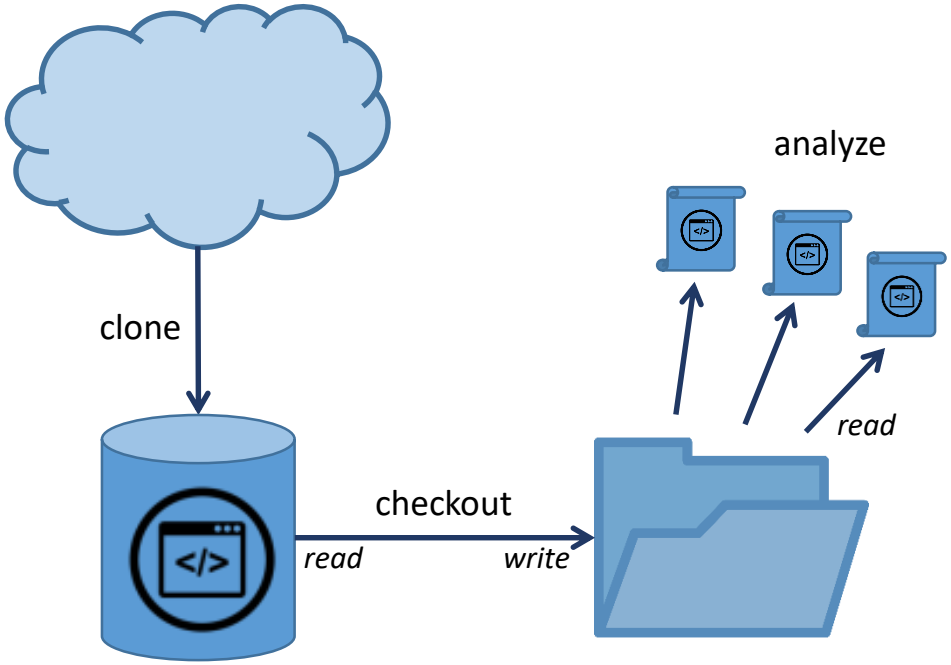
Avoid checkouts



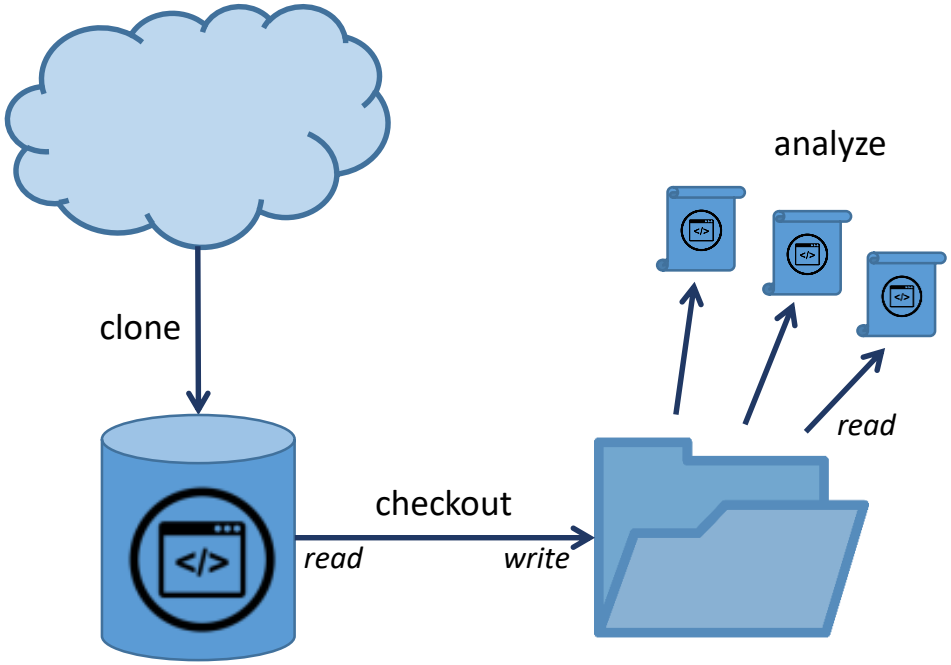
Avoid checkouts



Avoid checkouts

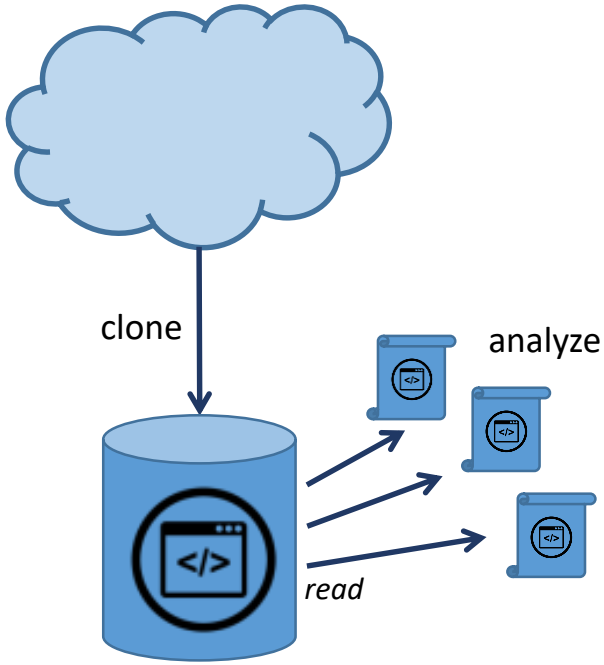


Avoid checkouts

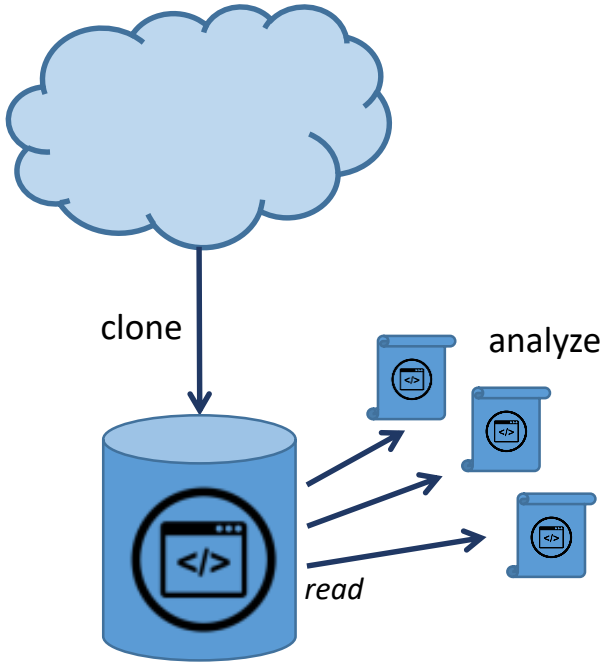


For every file: 2 read ops + 1 write op
Checkout includes irrelevant files
Need 1 CWD for every revision to be analyzed in parallel

Avoid checkouts

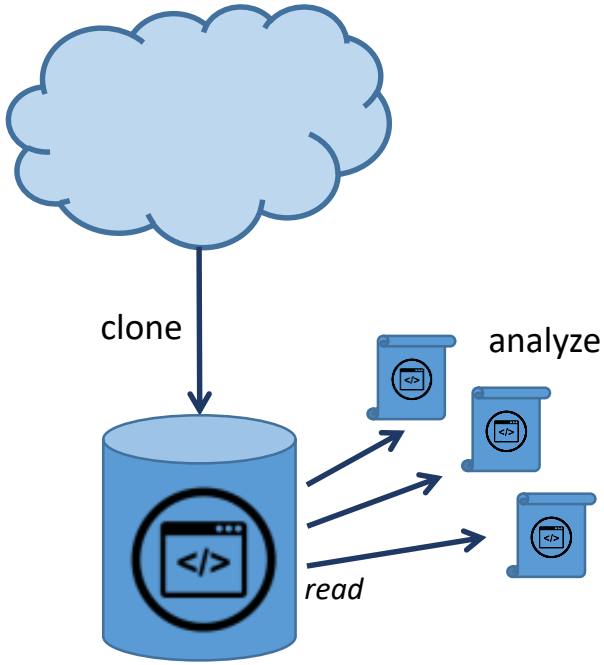


Avoid checkouts

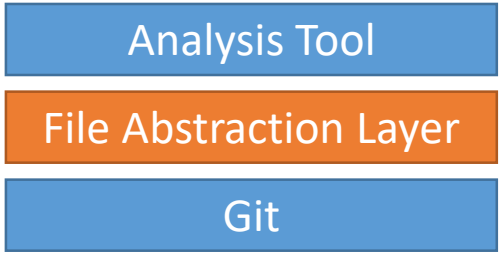


Only read relevant files in a single read op
No write ops
No overhead for parallization

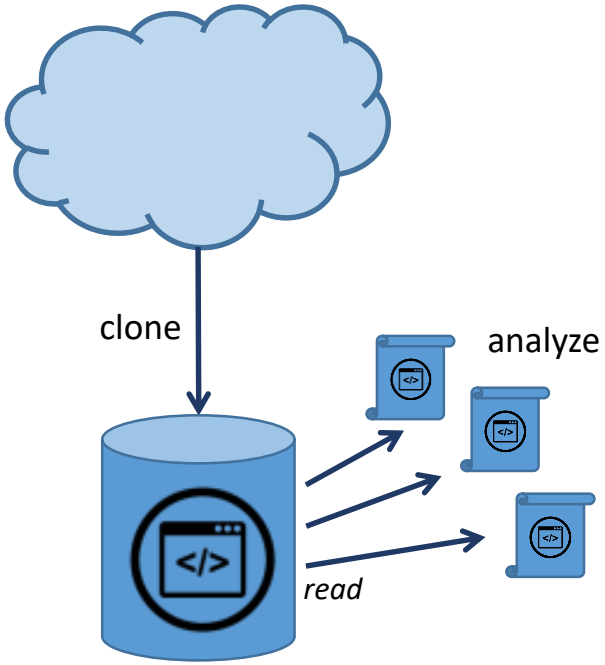
Avoid checkouts



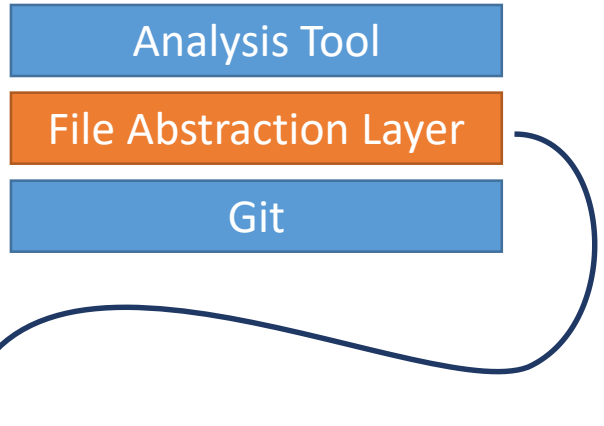
Only read relevant files in a single read op
No write ops
No overhead for parallization



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 

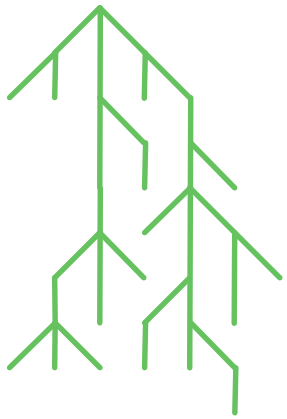
Merge ASTs

rev. 1

rev. 2

rev. 3

rev. 4



rev. 1

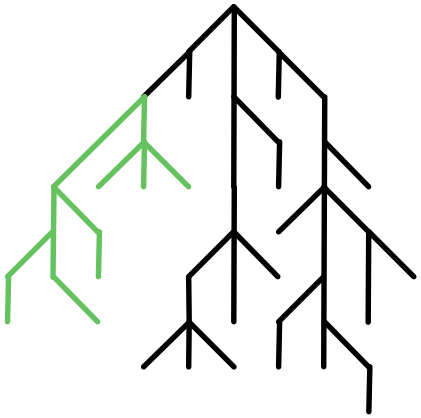
Merge ASTs

rev. 1

rev. 2

rev. 3

rev. 4



rev. 2

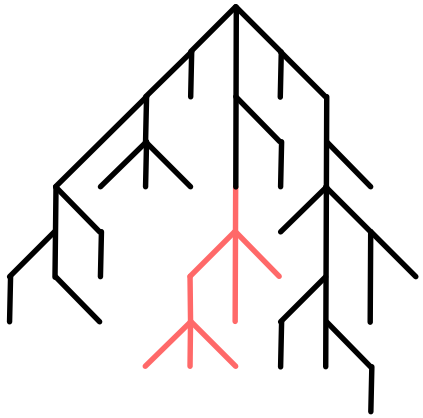
Merge ASTs

rev. 1

rev. 2

rev. 3

rev. 4



rev. 3

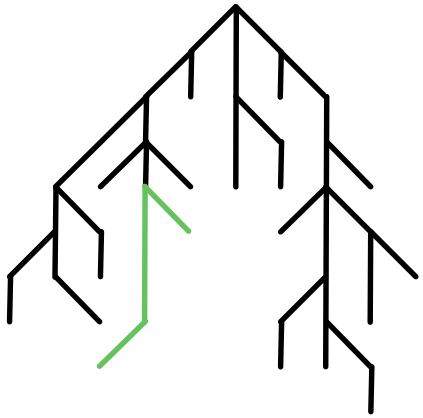
Merge ASTs

rev. 1

rev. 2

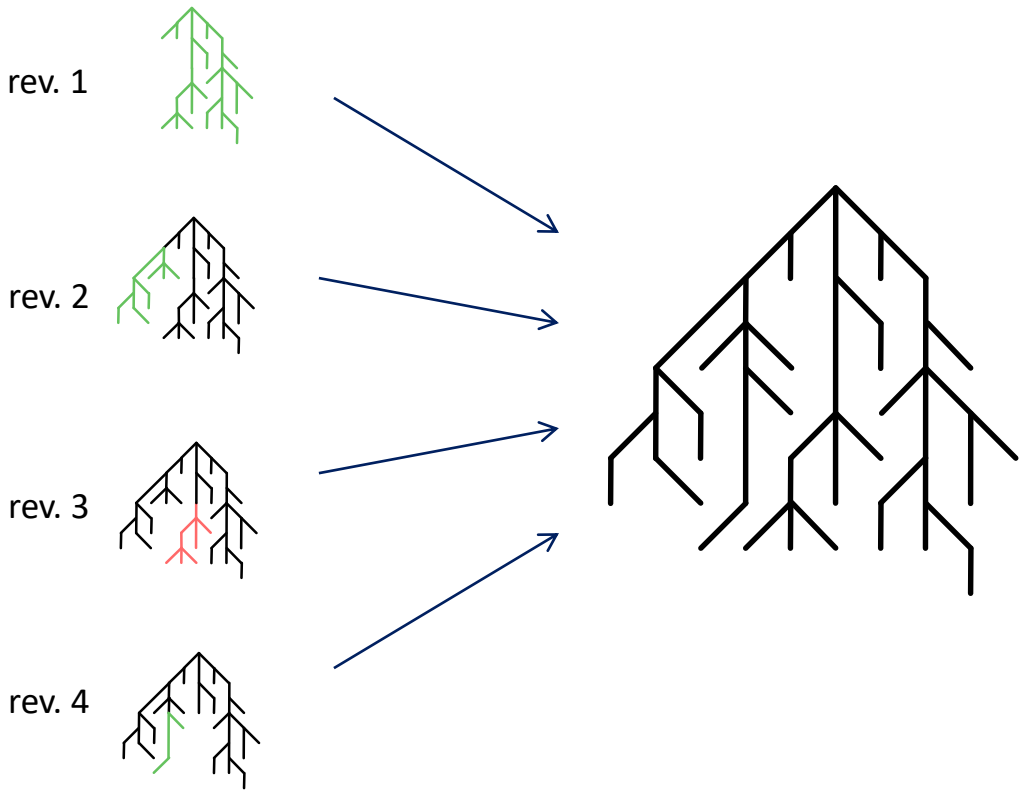
rev. 3

rev. 4

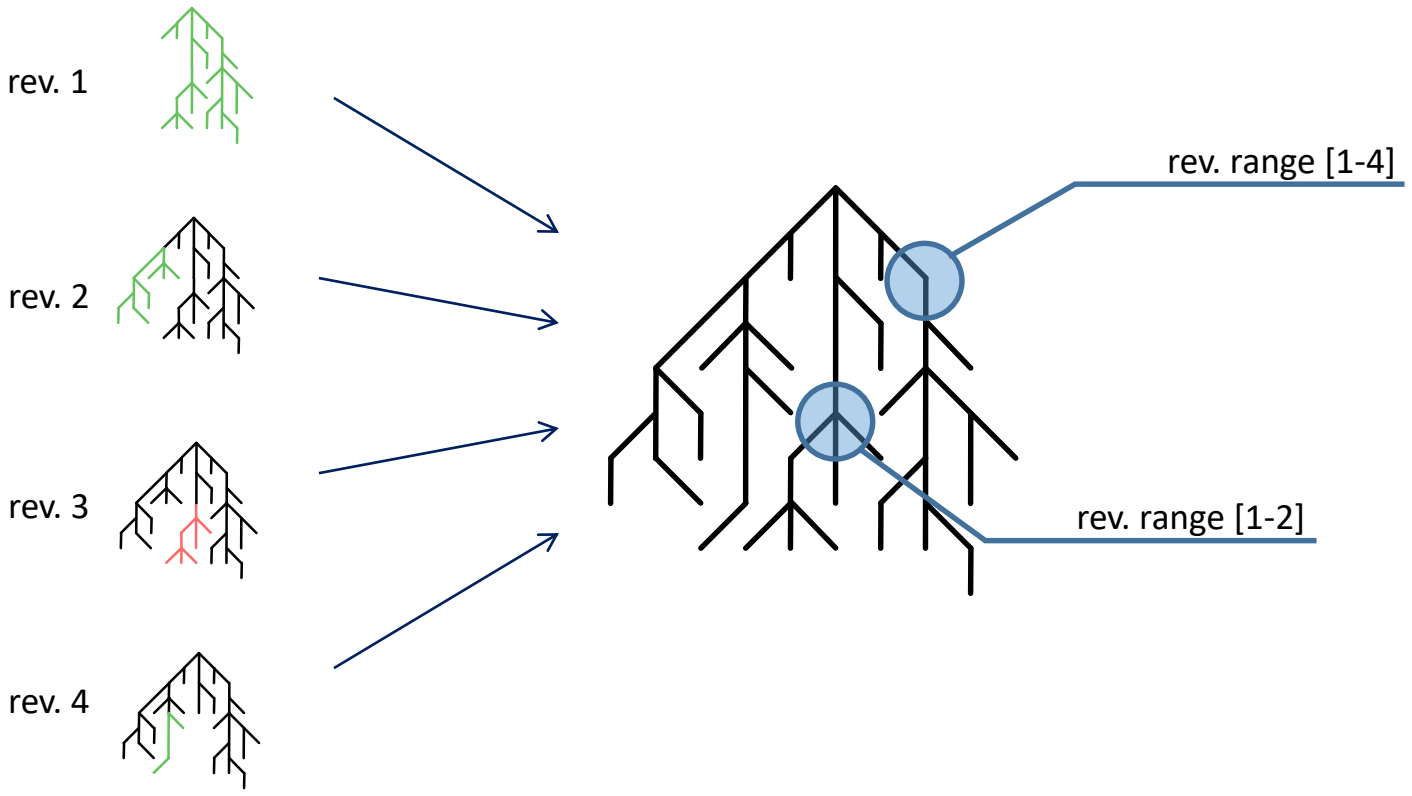


rev. 4

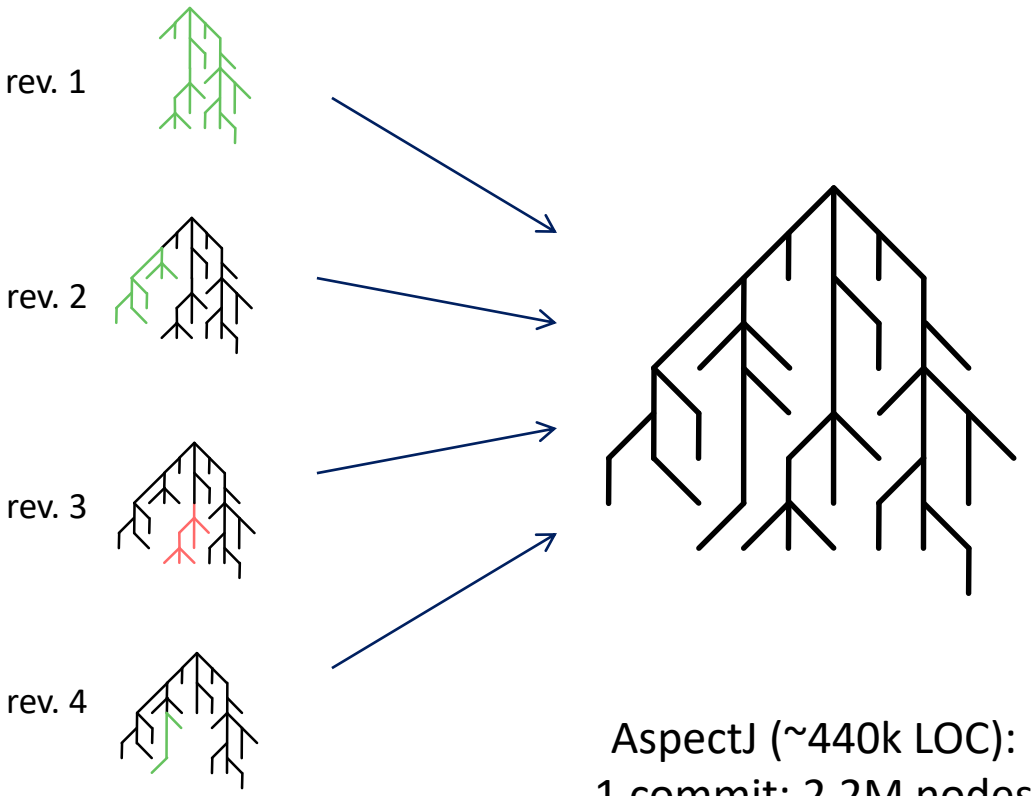
Merge ASTs



Merge ASTs



Merge ASTs



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

Merge ASTs

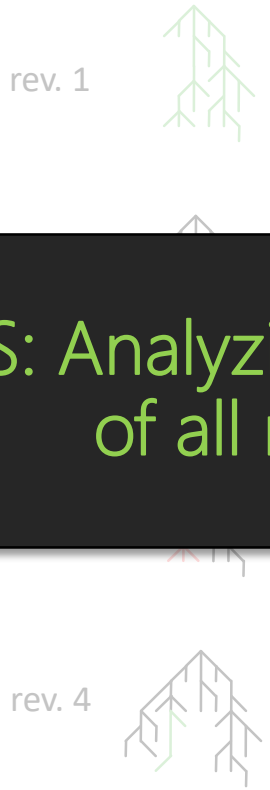
rev. 1

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

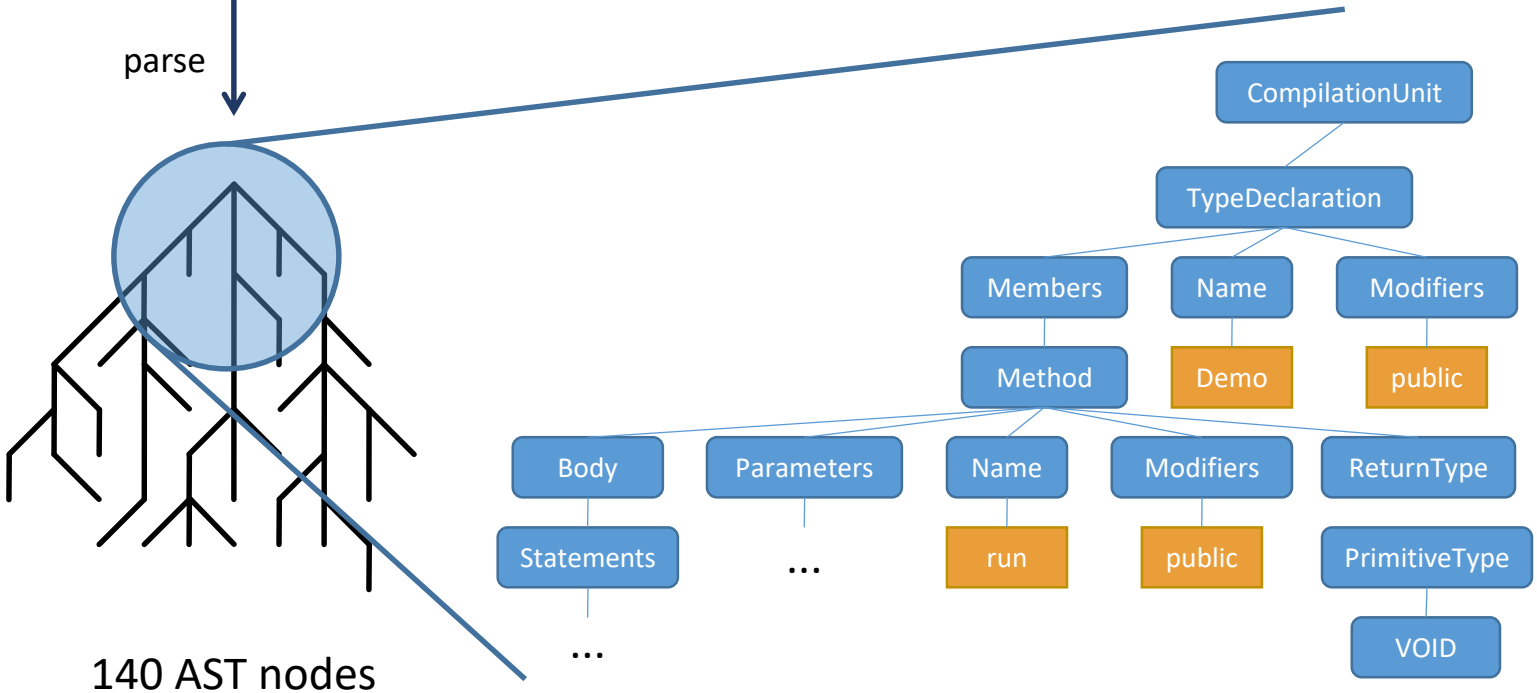
140 AST nodes
(using ANTLR)

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



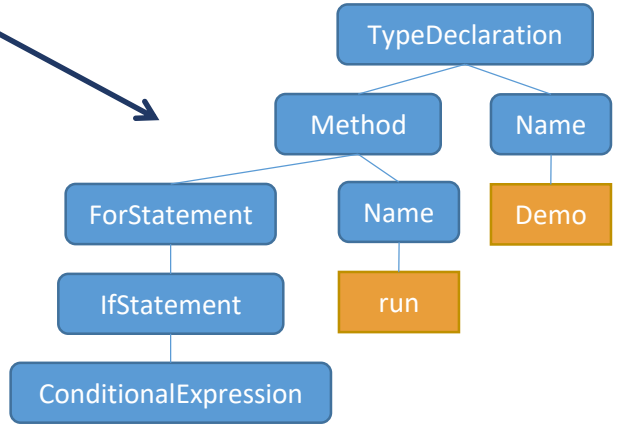
140 AST nodes
(using ANTLR)

```
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

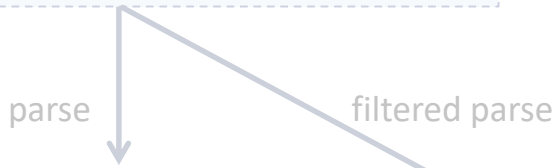


140 AST nodes
(using ANTLR)

7 AST nodes
(using ANTLR)

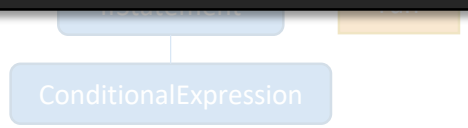
```
public class Demo {
  public void run() {
    for (int i = 1; i< 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
         System.out.println(i)
      }
  }
}
```

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

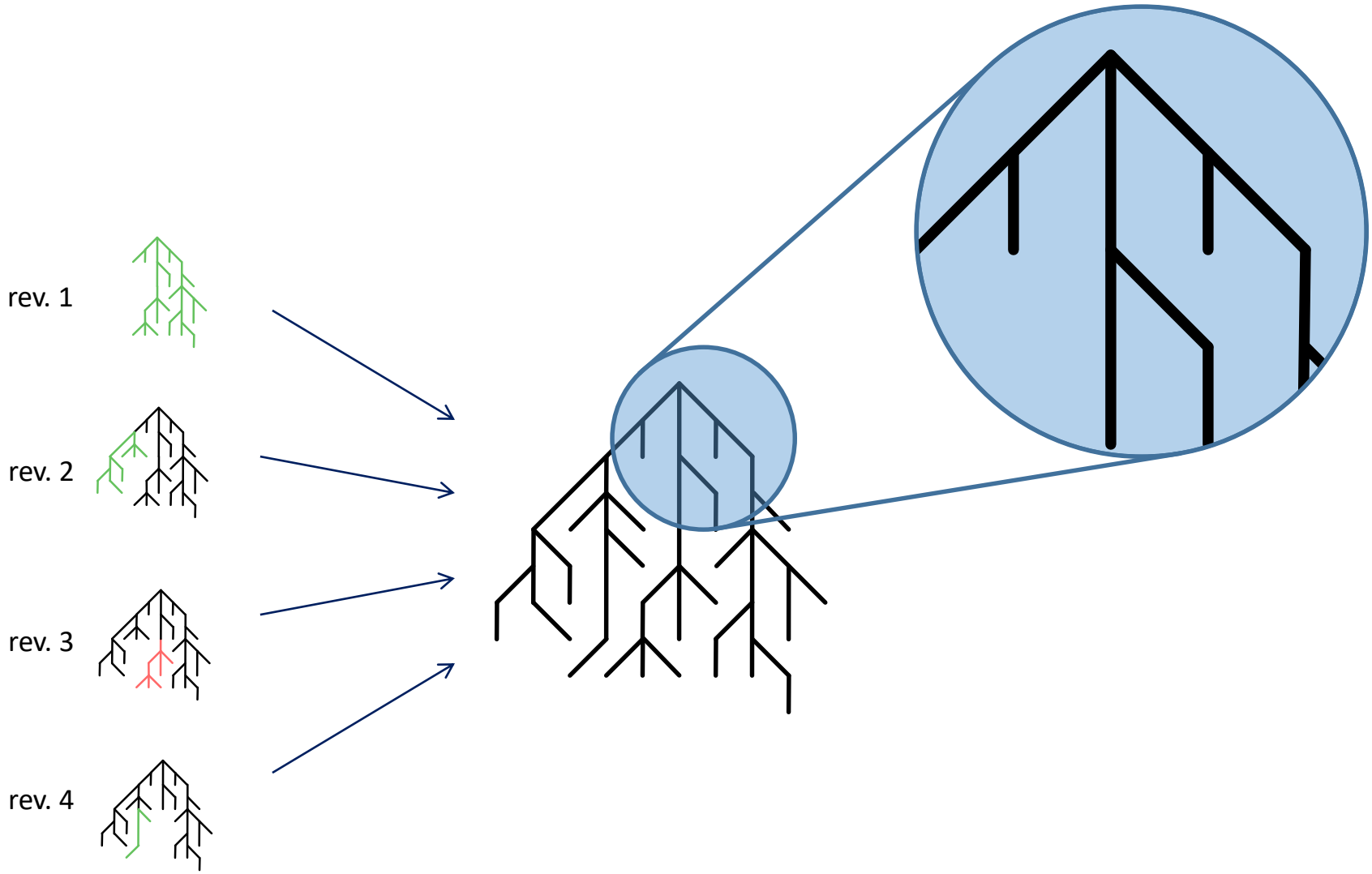
PS: Which AST nodes to load into the graph **depends on** the **analysis**

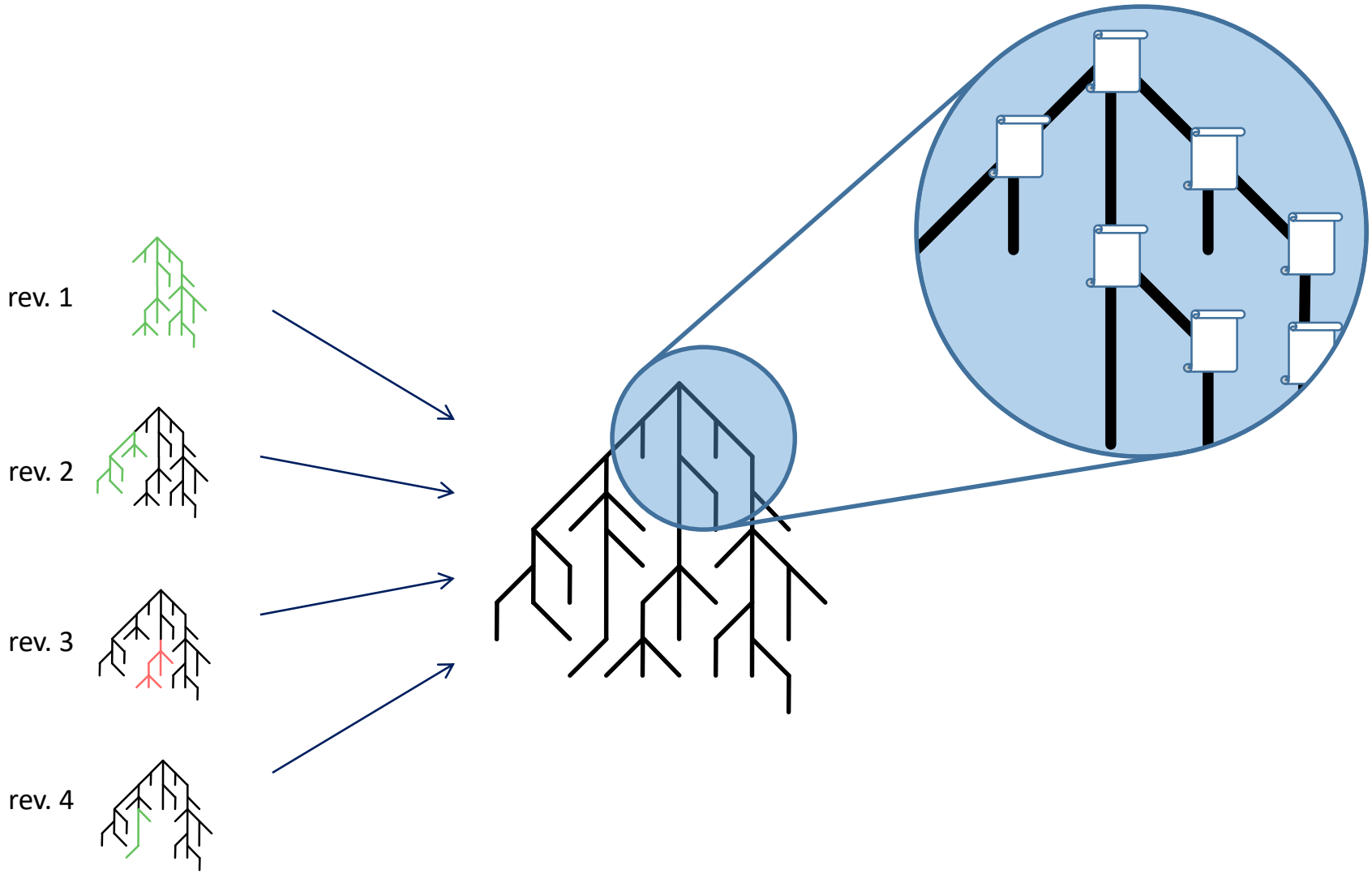
ConditionalExpression

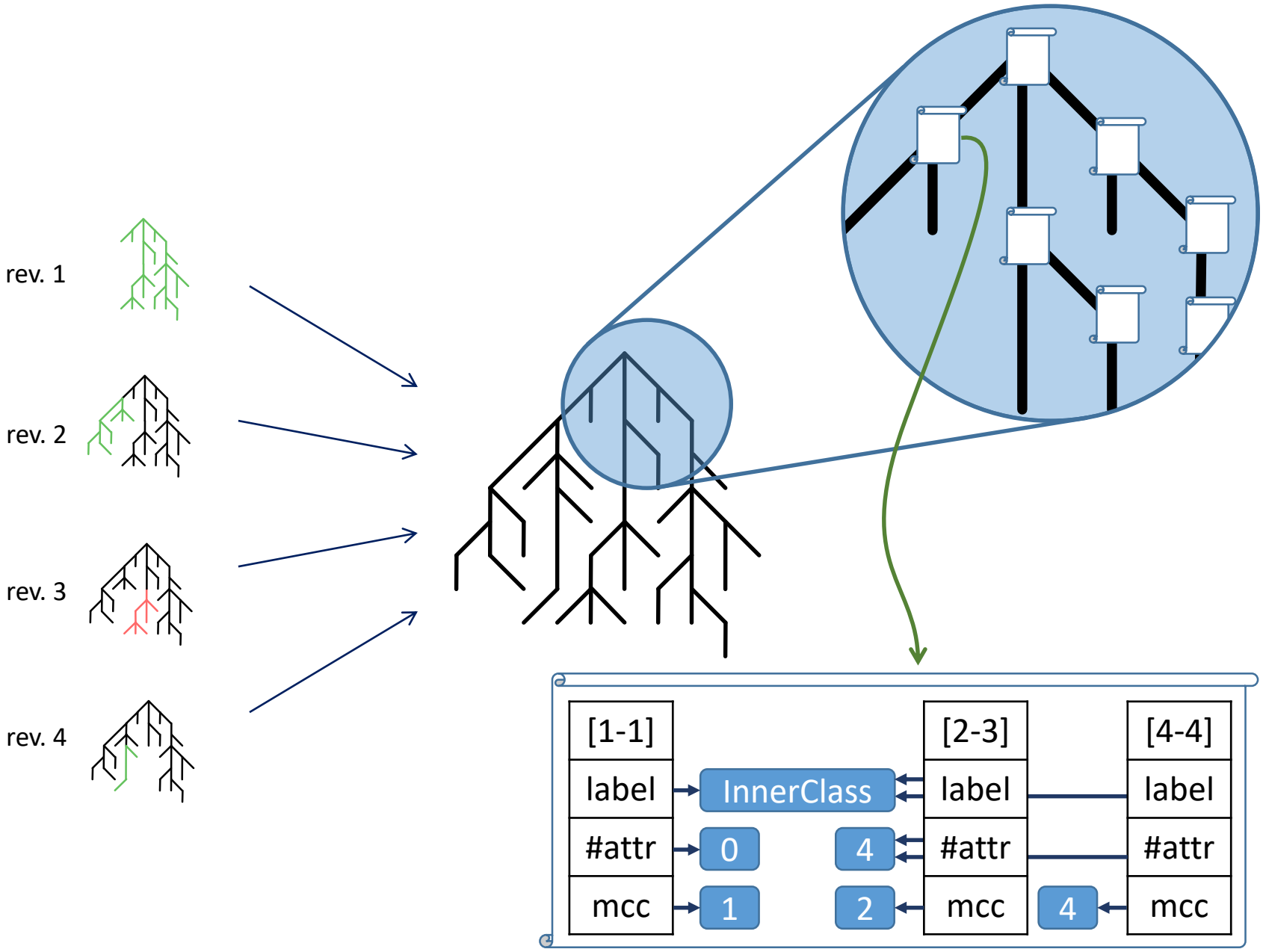
140 AST nodes
(using ANTLR)

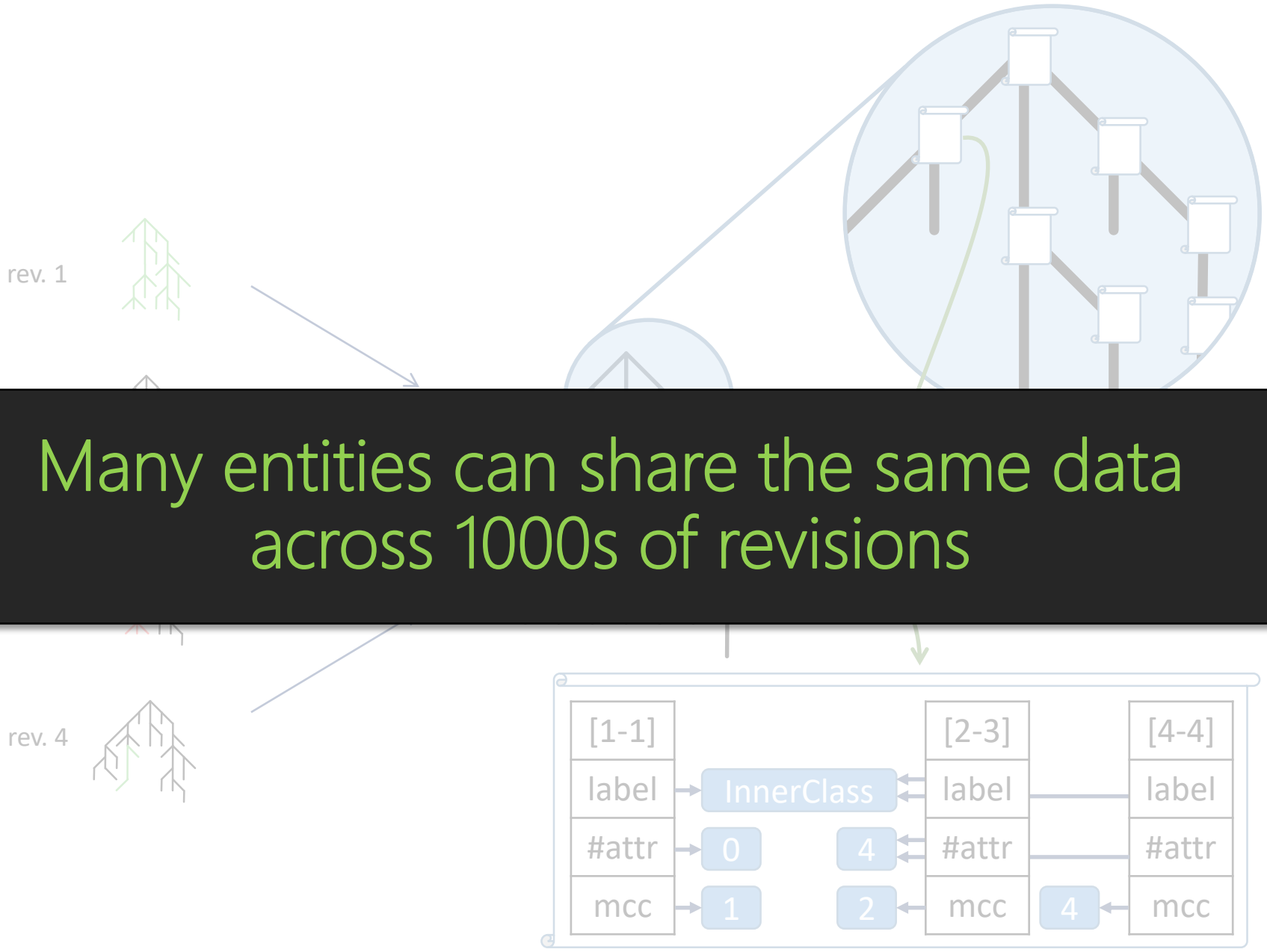
7 AST nodes
(using ANTLR)

#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**

A light-weight view on multi-language analysis

Typical solutions

- Toolchains / Frameworks
  - Integrate language-specific tooling
  - Lots of engineering required
- Meta-models
  - Translate language code to some common representation
  - Significant overhead / rigid models

Structure matters most

- Complexity?

```
if (true) {
  if (true) { }
} # CYCLO: 3
```

```
if (true) { }
if (true) { }
# CYCLO: 4
```

- # of Functions / Attributes etc.
- Coupling between Classes
- Call graphs

Relative structure is similar

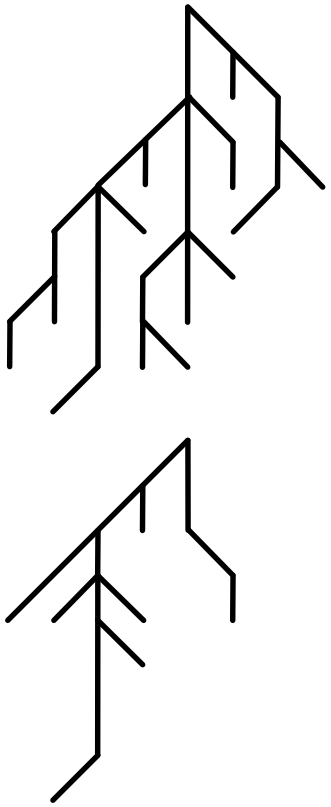
```
public class Demo {
  public void run() {
    for (int i = 1; i< 100; i++) {
      if (i % 3 == 0 || i % 5 == 0) {
         System.out.println(i)
}}}}
```

```
class Demo:
  def run():
    for i in range(1, 100):
      if i % 3 == 0 or i % 5 == 0:
        print(i)
```

Relative structure is similar

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

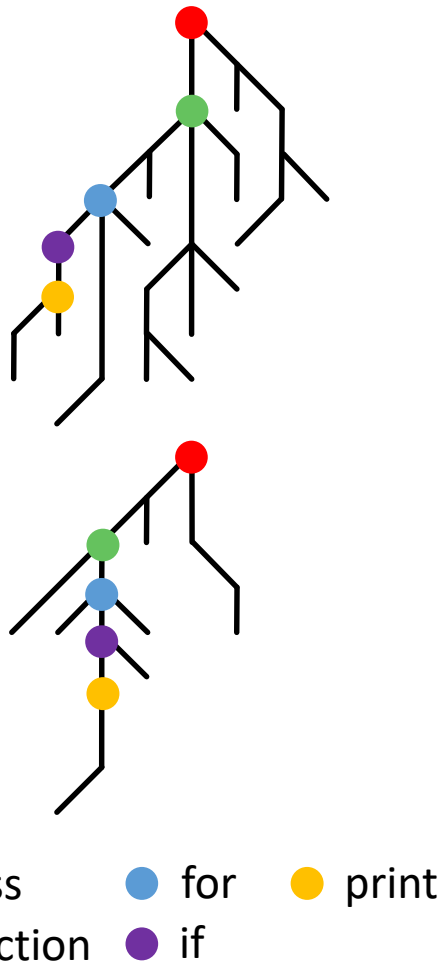
```
class Demo:
  def run():
    for i in range(1, 100):
      if i % 3 == 0 or i % 5 == 0:
        print(i)
```



Relative structure is similar

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

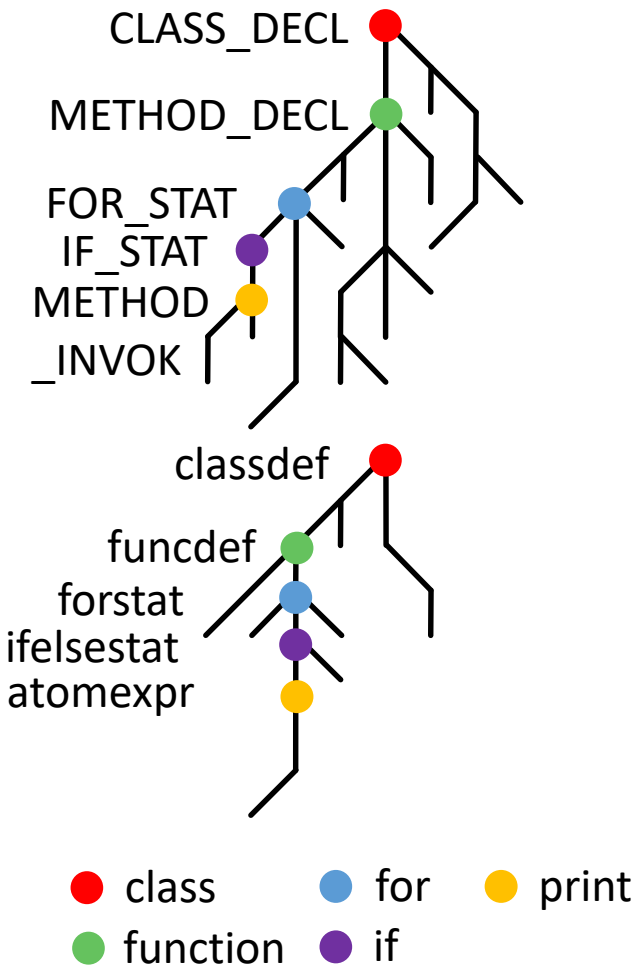
```
class Demo:
  def run():
    for i in range(1, 100):
      if i % 3 == 0 or i % 5 == 0:
        print(i)
```



Entities are different

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

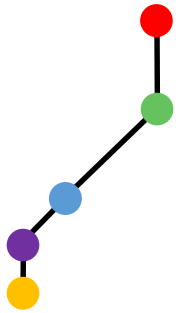
```
class Demo:
  def run():
    for i in range(1, 100):
      if i % 3 == 0 or i % 5 == 0:
        print(i)
```

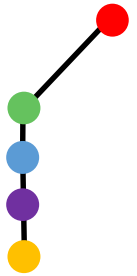


Can filter irrelevant nodes

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

```
class Demo:
  def run():
    for i in range(1, 100):
      if i % 3 == 0 or i % 5 == 0:
        print(i)
```

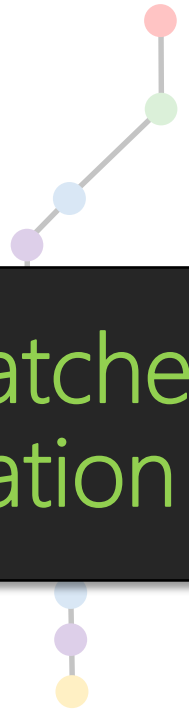




class for print
function if

Can filter irrelevant nodes

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



View: Structure already matches, only entities need translation

```
  def run():
    for i in range(1, 100):
      if i % 3 == 0 or i % 5 == 0:
        print(i)
```

class for print
function if

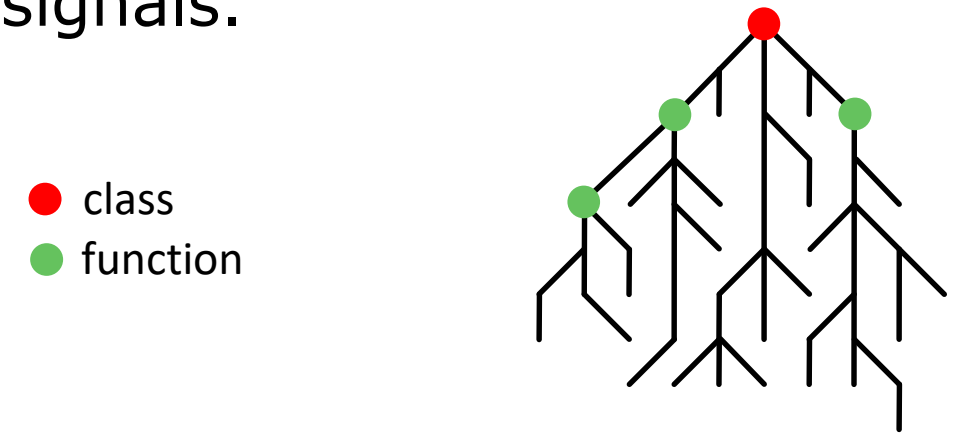
How LISA handles Multiple languages

Analysis formulation

- Signal/Collect (like "Google Pregel" for Scala)
  - Graph vertices send information packets (signals) and do something when receiving (collecting) signals.

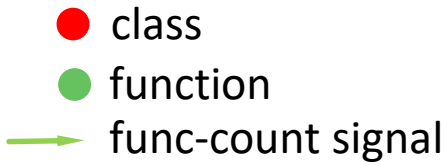
Analysis formulation

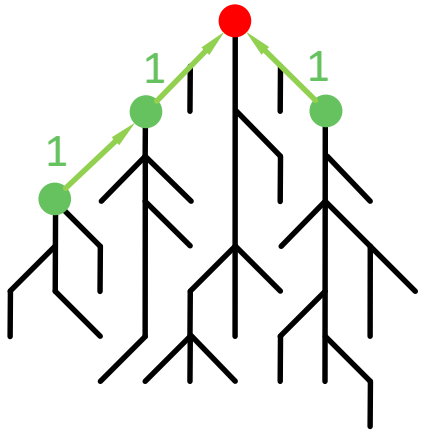
- Signal/Collect (like "Google Pregel" for Scala)
  - Graph vertices send information packets (signals) and do something when receiving (collecting) signals.



Analysis formulation

- Signal/Collect (like "Google Pregel" for Scala)
  - Graph vertices send information packets (signals) and do something when receiving (collecting) signals.

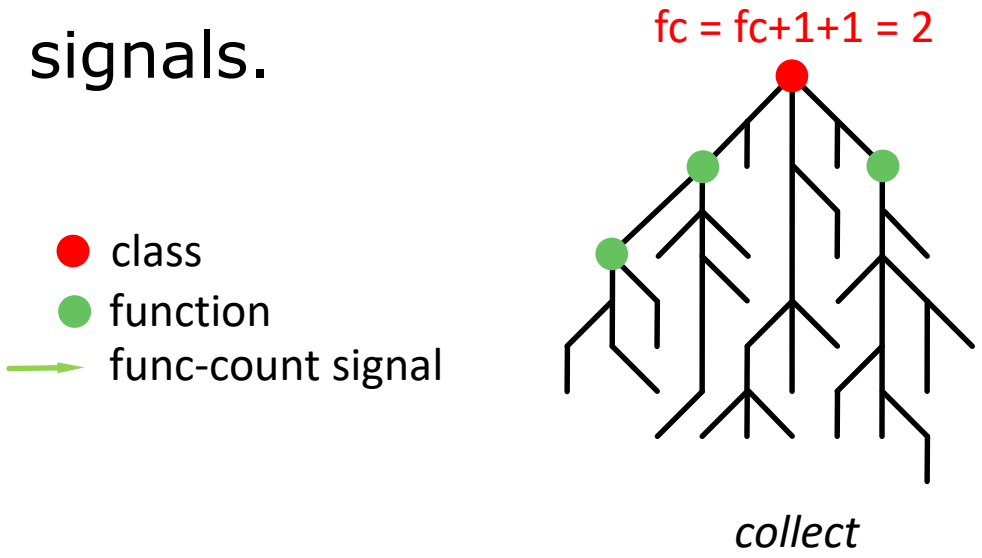




signal

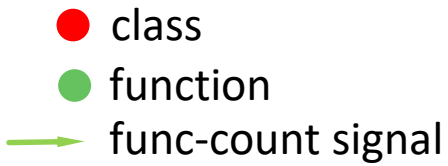
Analysis formulation

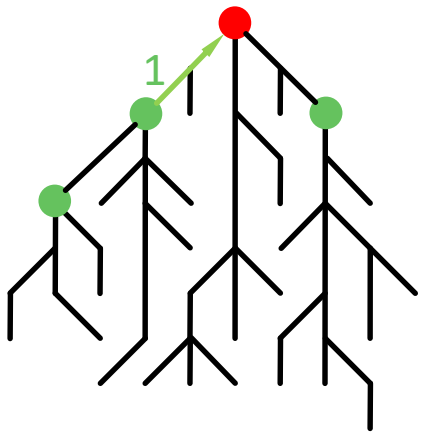
- Signal/Collect (like "Google Pregel" for Scala)
  - Graph vertices send information packets (signals) and do something when receiving (collecting) signals.



Analysis formulation

- Signal/Collect (like "Google Pregel" for Scala)
  - Graph vertices send information packets (signals) and do something when receiving (collecting) signals.

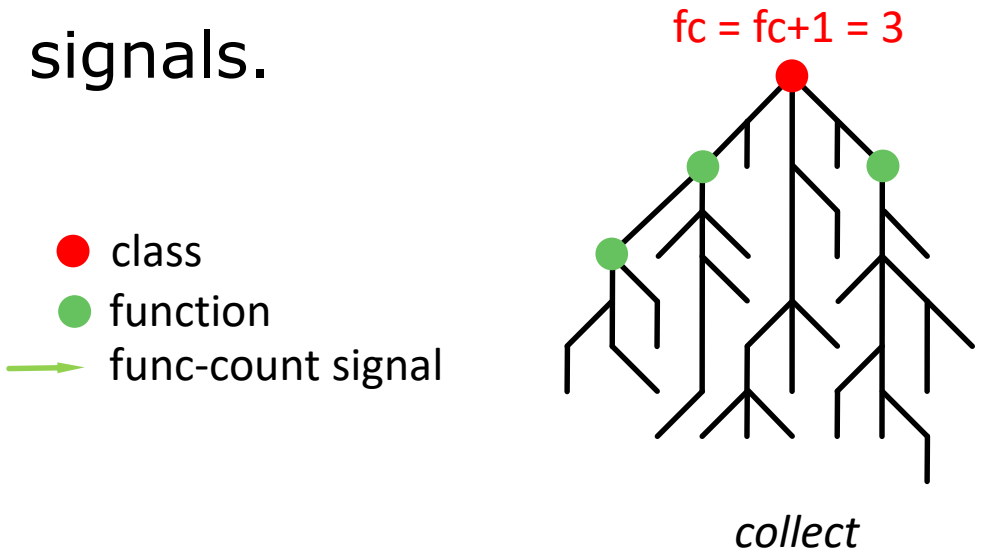




signal

Analysis formulation

- Signal/Collect (like "Google Pregel" for Scala)
  - Graph vertices send information packets (signals) and do something when receiving (collecting) signals.



Analysis formulation

- Signal/Collect (like "Google Pregel" for Scala)
  - Graph vertices send information packets (signals) and do something when receiving (collecting) signals.
- Analyses use *generic lables* for entities

```
override def start = {·
implicit domain => state =>
  if (leaf(state))
    state ! new MccPacket(
      if (state is 'branch) 2 else 1)
  else state
}

class MccPacket(val count: Int)
extends AnalysisPacket {
  override def onCollect = {
  implicit domain => state =>

    val mcc = state[Mcc].n + count - 1

    allChildren[Mcc](state)(
      incomplete = state + Mcc(false, mcc),
      complete = {
        val newCount =·
          if (state is 'branch) mcc + 1 else mcc
        val persist =
          state is ('class, 'method, 'unit)
        state + Mcc(persist, mcc)
              ! new MccPacket(newCount) })}}}
```

```
override def start = {·
implicit domain => state =>
  if (leaf(state))
    state ! new MccPacket(
      if (state is 'branch) 2 else 1)
  else state
}

class MccPacket(val count: Int)
extends AnalysisPacket {
  override def onCollect = {
  implicit domain => state =>

    val mcc = state[Mcc].n + count - 1

    allChildren[Mcc](state)(
      incomplete = state + Mcc(false, mcc),
      complete = {
        val newCount =·
          if (state is 'branch) mcc + 1 else mcc
        val persist =
          state is ('class, 'method, 'unit)
        state + Mcc(persist, mcc)
              ! new MccPacket(newCount) })}}}
```

Light-weight entity mappings

```
object PythonNativeParseTree extends Domain {
  override val mapping = Map(
    'file       -> Set("META-FILE"),
    'class      -> Set("ClassDefbody"),
    'method     -> Set("FunctionDefbody"),
    'name       -> Set("Name"),
    'idiomatic -> Set("ListComp", "GeneratorExpargs", "Lambdaargs",
                      "Yield", "Exprfinalbody", "Withbody",
                      "Namedecorator_list"),
    'fork       -> Set("Whilebody", "Forbody", "Ifbody", "Trybody",
                      "Withbody", "ListComp", "SetComp", "BoolOp",
                      "IfExp"),
    'block      -> Set("Whilebody", "Forbody", "Ifbody", "Trybody",
                      "Withbody", "ClassDefbody", "FunctionDefbody"),
    'variable   -> Set("Assignbody", "AugAssignbody"),
    'parameter -> Set("argargs")
  )
}
```

Light-weight entity mappings

```
object PythonNativeParseTree extends Domain {
  override val mapping = Map(
    'file       -> Set("META-FILE"),
    'class      -> Set("ClassDefbody"),
    'method     -> Set("FunctionDefbody"),
    'name       -> Set("Name"),
    'idiomatic  -> Set("ListComp", "GeneratorExpargs", "Lambdaargs",
                       "Yield", "Exprfinalbody", "Withbody",
                       ),
    'fork       -> Set(                    y", "Ifbody", "Trybody",
                       "Withbody", "ListComp", "SetComp", "BoolOp",
                       "IfExp"),
    'block      -> Set("Whilebody", "Forbody", "Ifbody", "Trybody",
                       "Withbody", "ClassDefbody", "FunctionDefbody"),
    'variable   -> Set("Assignbody", "AugAssignbody"),
    'parameter  -> Set("argargs")
  )
}
```

Lables used by Analyses

Light-weight entity mappings

```
object PythonNativeParseTree extends Domain {
  override val mapping = Map(
    'file      -> Set("META-FILE"),
    'class     -> Set("ClassDefbody"),
    'method    -> Set("FunctionDefbody"),
    'name      -> Set("Name"),
    'lambda    -> Set("ListComp", "GeneratorExpargs", "Lambdaargs",
                      "Yield", "Exprfinalbody", "Withbody",
                      "Namedecorator_list"),
    'expr      -> Set("Whilebody", "Forbody", "Ifbody", "Trybody",
                      "Withbody", "ListComp", "SetComp", "BoolOp",
                      "IfExp"),
    'block     -> Set("Whilebody", "Forbody", "Ifbody", "Trybody",
                      "Withbody", "ClassDefbody", "FunctionDefbody"),
    'variable  -> Set("Assignbody", "AugAssignbody"),
    'parameter -> Set("argargs")
  )
}
```

Lables used by the parser or grammar

Light-weight entity mappings

```
object PythonNativeParseTree extends Domain {
  override val mapping = Map(
    'file       -> Set("META-FILE"),
    'class      -> Set("ClassDefbody"),
    'method     -> Set("FunctionDefbody"),
    'name       -> Set("Name"),
    'idiomatic -> Set("ListComp", "GeneratorExpargs", "Lambdaargs",
                      "Yield", "Exprfinalbody", "Withbody",
                      "Namedecorator_list"),
    'fork       -> Set("Whilebody", "Forbody", "Ifbody", "Trybody",
                      "Withbody", "ListComp", "SetComp", "BoolOp",
                      "IfExp"),
    'block      -> Set("Whilebody", "Forbody", "Ifbody", "Trybody",
                      "Withbody", "ClassDefbody", "FunctionDefbody"),
    'variable   -> Set("Assignbody", "AugAssignbody"),
    'parameter -> Set("argargs")
  )
}
```

Parsing

- AST is kept as the parser supplies it
- ANTLRv4 integration
- Filtering can be enabled
  - Only AST nodes that correspond to a label used in an analysis are kept
  - Reduces graph size by a factor of 10 or more

Adding new languages

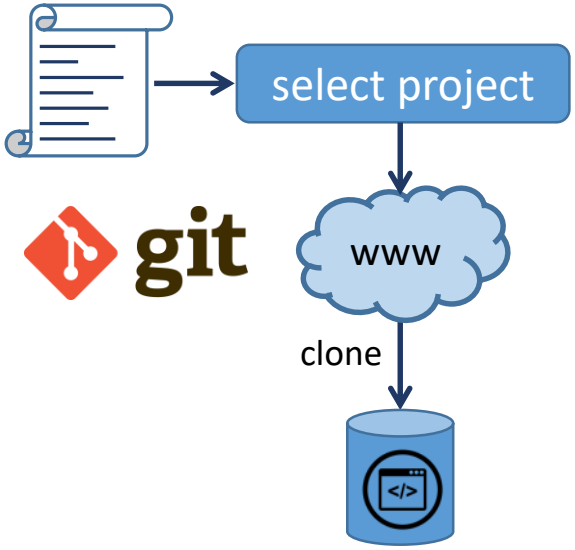
1. Integrate a parser (or generate one)
   - Graph interface allows adding nodes/edges
2. Write a node mapping
3. Re-use existing analyses on new ASTs

```
package org.example
import ch.uzh.ifi.seal.lisa.core._
import ch.uzh.ifi.seal.lisa.core.public._
import ch.uzh.ifi.seal.lisa.antlr._
import ch.uzh.ifi.seal.lisa.module.analysis._
import ch.uzh.ifi.seal.lisa.module.persistence.CSVPersistence

import org.example.parser.LuaLexer
import org.example.parser.LuaParser

object AntlrLuaParseTree extends Domain {
  override val mapping = Map(
    'file       -> Set("Chunk"),
    'block      -> Set("Block"),
    'statement  -> Set("Stat"),
    'fork       -> Set("DoStat", "WhileStat", "RepeatStat", "IfStat", "ForStat", "ForInStat"),
    'method     -> Set("Function"),
    'variable   -> Set("Var"),
    'field      -> Set("Field")
  )
}

object AntlrLuaParser extends AntlrParser[LuaParser](AntlrLuaParseTree) {
  override val suffixes = List(".lua")
  override def lex(input: ANTLRInputStream) = new LuaLexer(input)
  override def parse(tokens: CommonTokenStream) = new LuaParser(tokens)
  override def enter(parser: LuaParser) = parser.chunk()
}
```

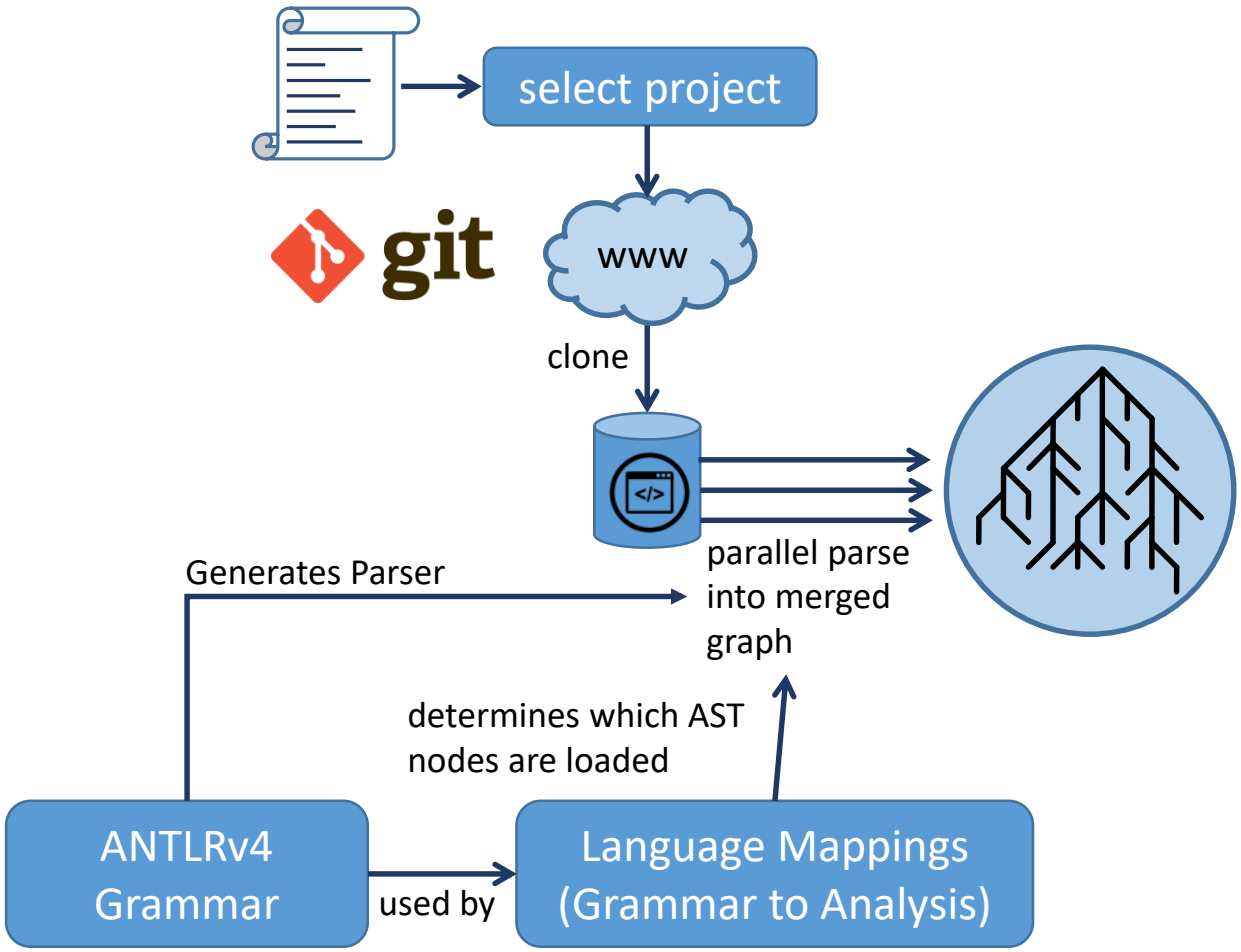
```
object LuaAnalysis extends App {
  val url = "https://github.com/Mashape/kong.git"
  implicit val uid = "Mashape_kong"
  val gitLocalDir = s"/tmp/lisa/git/$uid"
  val targetDir = s"/tmp/lisa/results/$uid"
  val parsers = List[Parser](AntlrLuaParser)
  val analyses = UniversalAnalysisSuite
  val persistence = new CSVPersistence(targetDir)
  val sources = new GitAgent(parsers, url, gitLocalDir)
  val c = new LisaComputation(sources, analyses, persistence)
  c.execute
}
```

To Summarize...

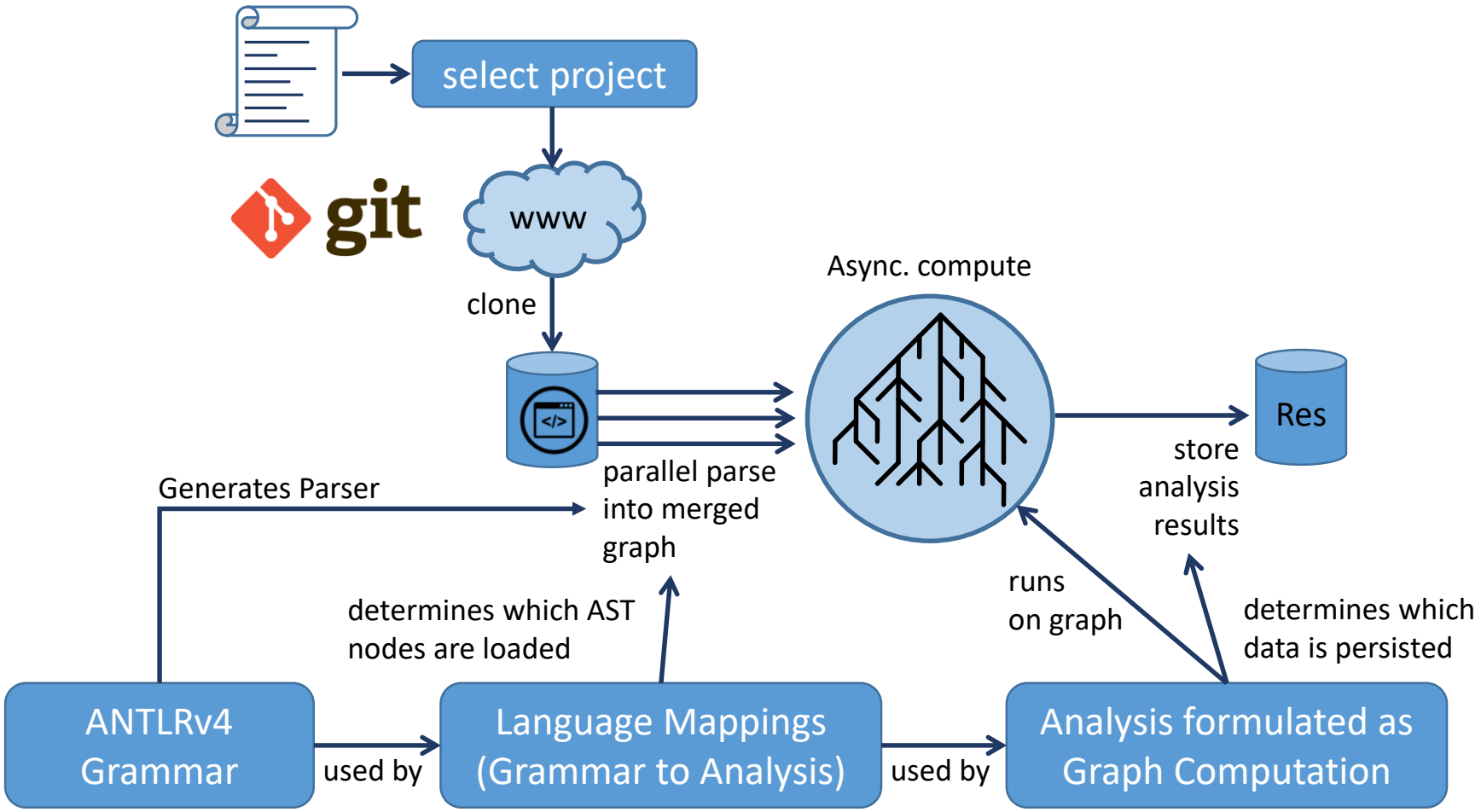
The LISA Analysis Process



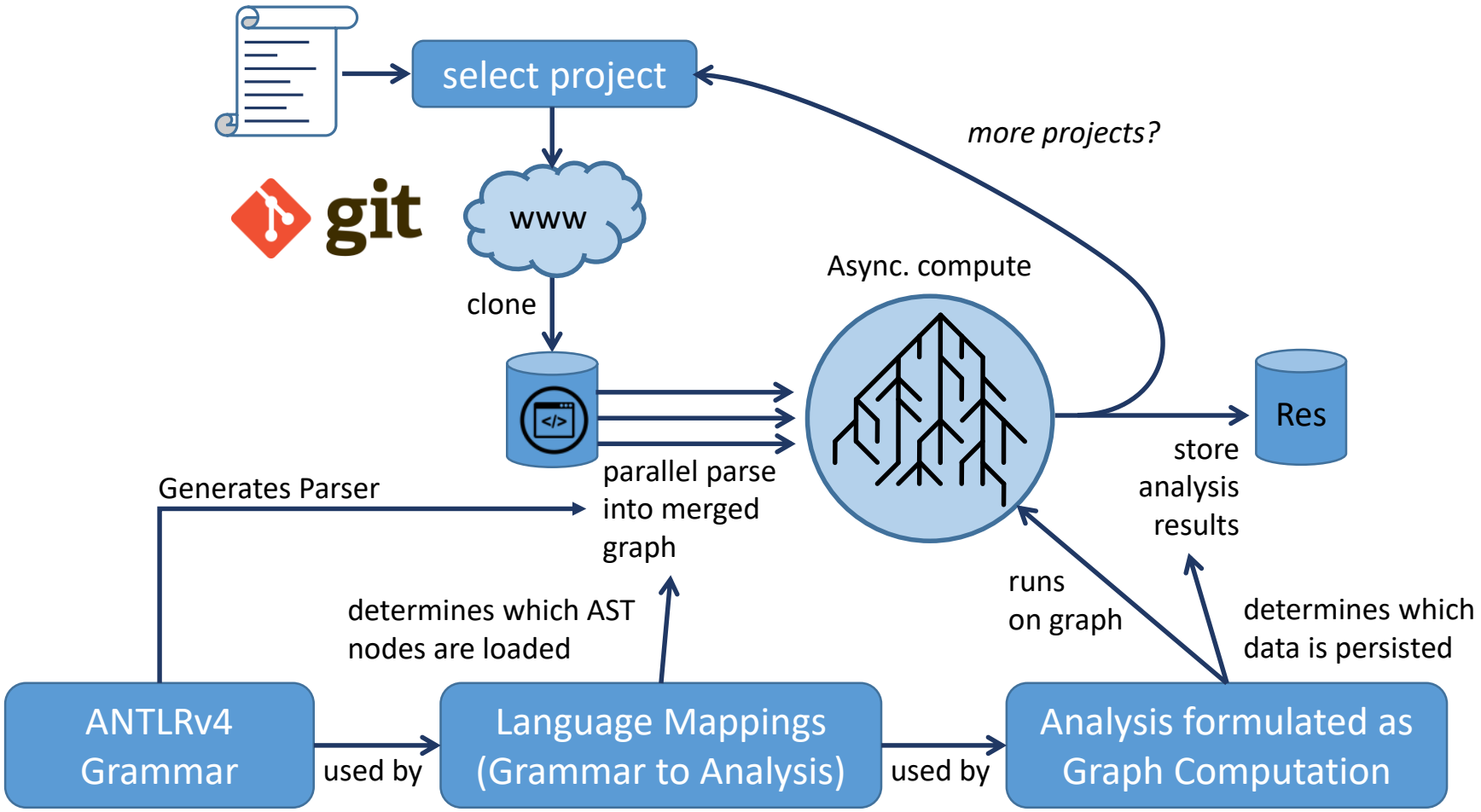
The LISA Analysis Process



The LISA Analysis Process



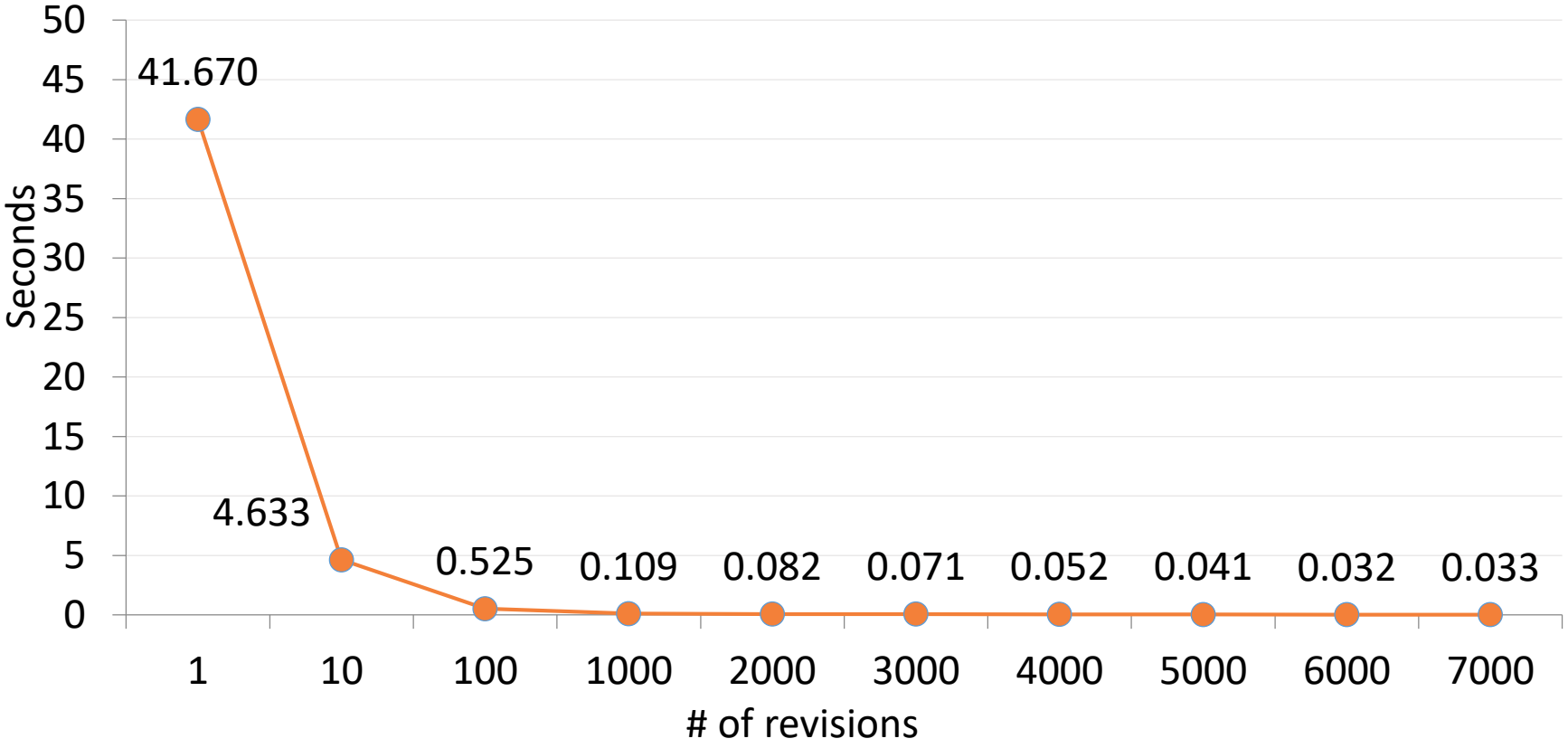
The LISA Analysis Process



How well does it work, then?

Marginal cost for +1 revision

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



Overall Performance Stats

Language	Java	C#	JavScript	Python
#Projects	1000	1000	1000	1000
#Revisions	2 Million	1.4 Million	1.5 Million	2.3 Million
#Files (parsed!)	10 Billion	3 Billion	380 Thousand	1.2 Billion
#Lines (parsed!)	1.6 Trillion	0.6 Trillion	43 Billion	0.3 Trillion
Total Runtime (RT)[1]	2d 5h	3d	23h	2d 4h
Median RT[1]	8.35s	40.5s	14.4s	24.5s
Tot. Avg. RT per Rev.[2]	97ms	183ms	57ms	83ms
Med. Avg. RT per Rev.[2]	**41ms**	**88ms**	**25ms**	**48ms**

[1] Including cloning and persisting results
[2] Excluding cloning and persisting results

What's the catch?

(There are a few...)

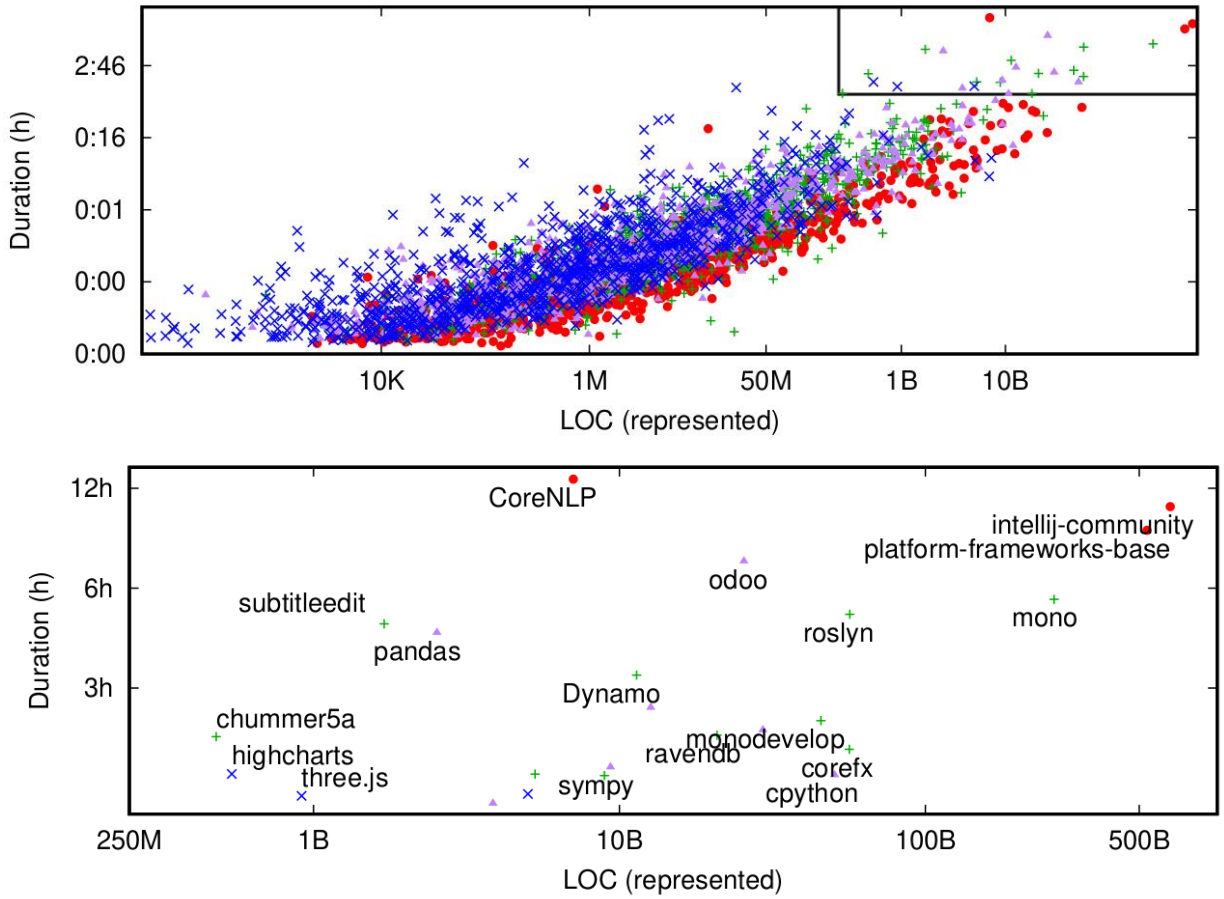
The (not so) minor stuff

- 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)

Parser performance matters



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

What's next?

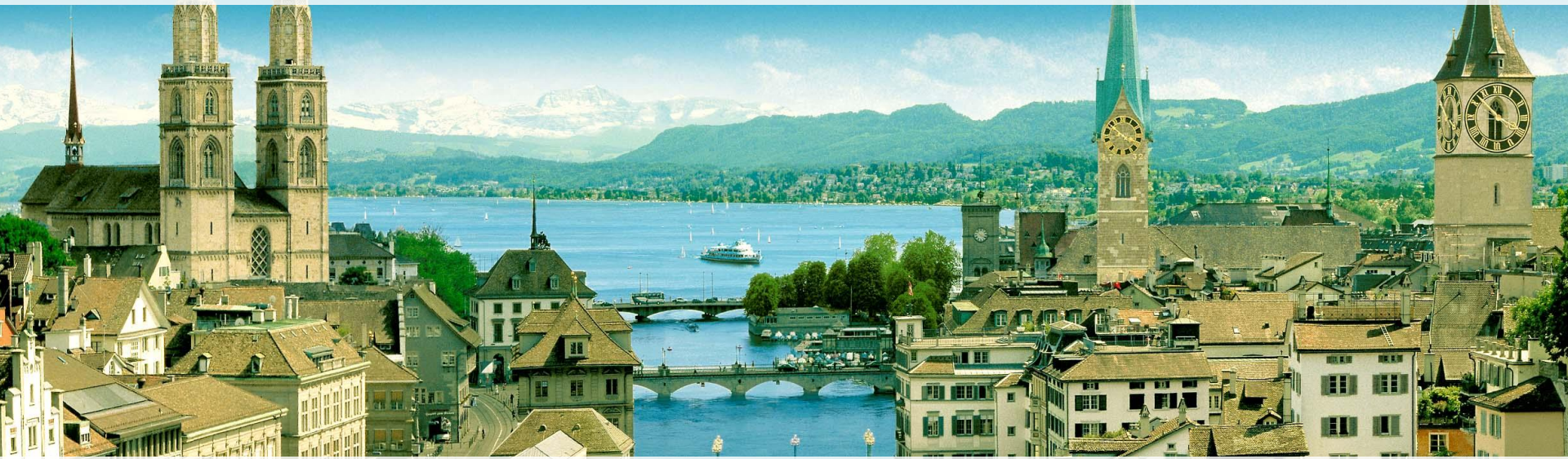
- LISA for Java Bytecode:
  - Use ASM to load class data into LISA

What's next?

- LISA for Java Bytecode:
  - Use ASM to load class data into LISA
  - Makes it easier to do multi-file analyses
    - Coupling, inheritance, refactorings…

What's next?

- LISA for Java Bytecode:
  - Use ASM to load class data into LISA
  - Makes it easier to do multi-file analyses
    - Coupling, inheritance, refactorings…
  - Use existing, pre-compiled snapshots
    - Archived nightly builds
    - Related work on historical compilation



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

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

Get the slides: http://t.uzh.ch/S4

Contact me: alexandru@ifi.uzh.ch