

Scala ⁱⁿ Practice

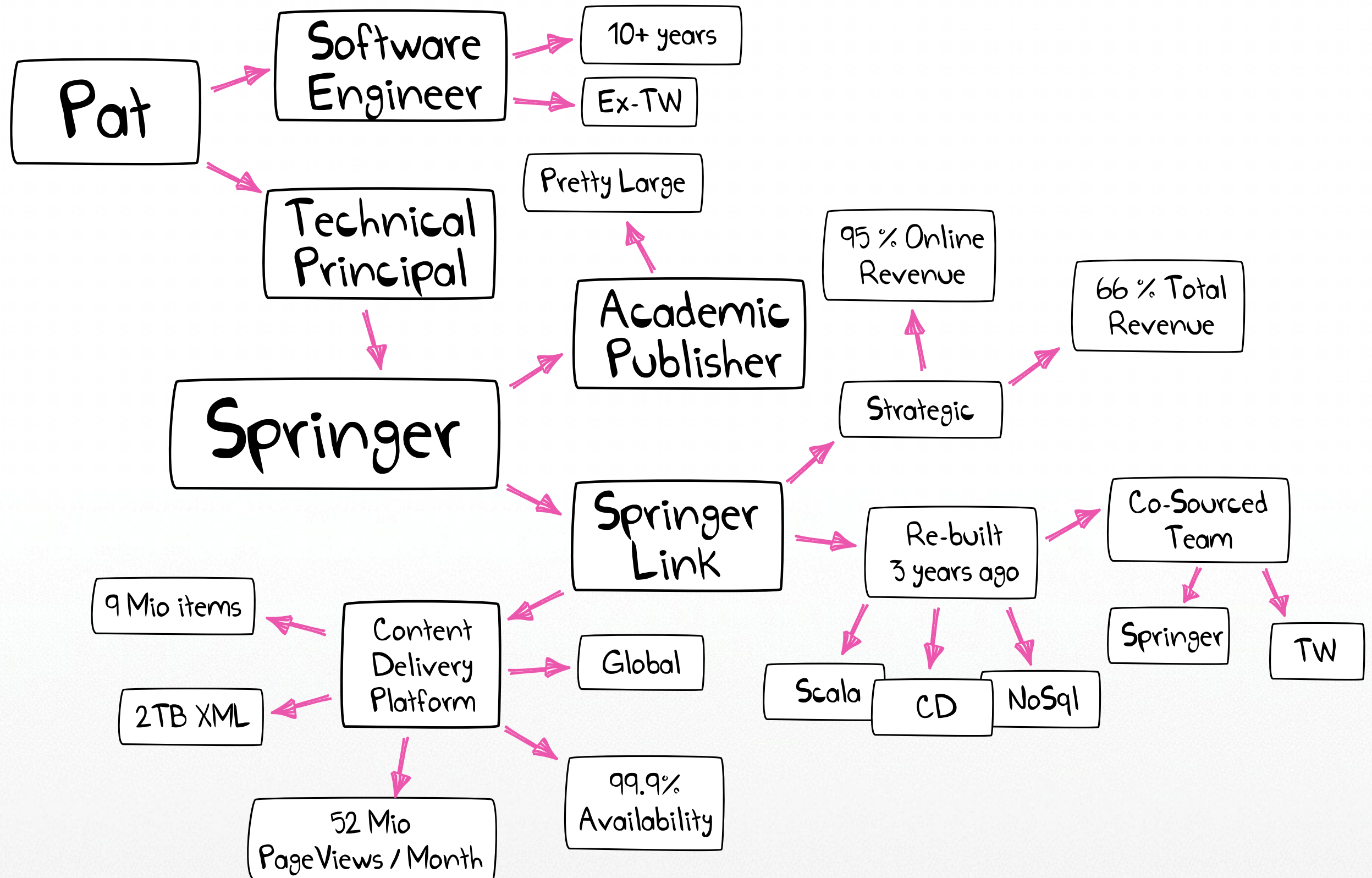
3 years later...

@patforna

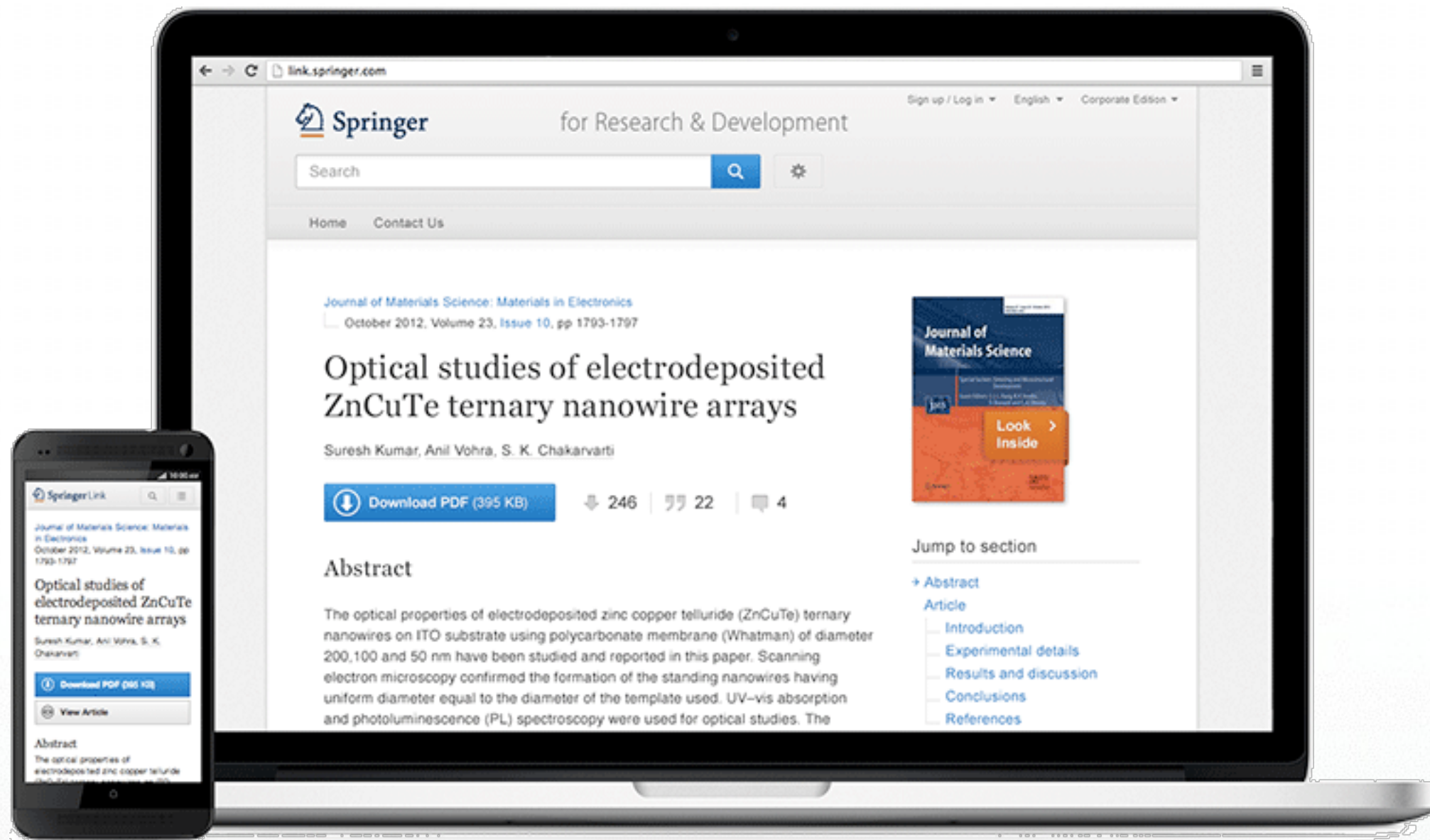
patric.fornasier@springer.com

Context

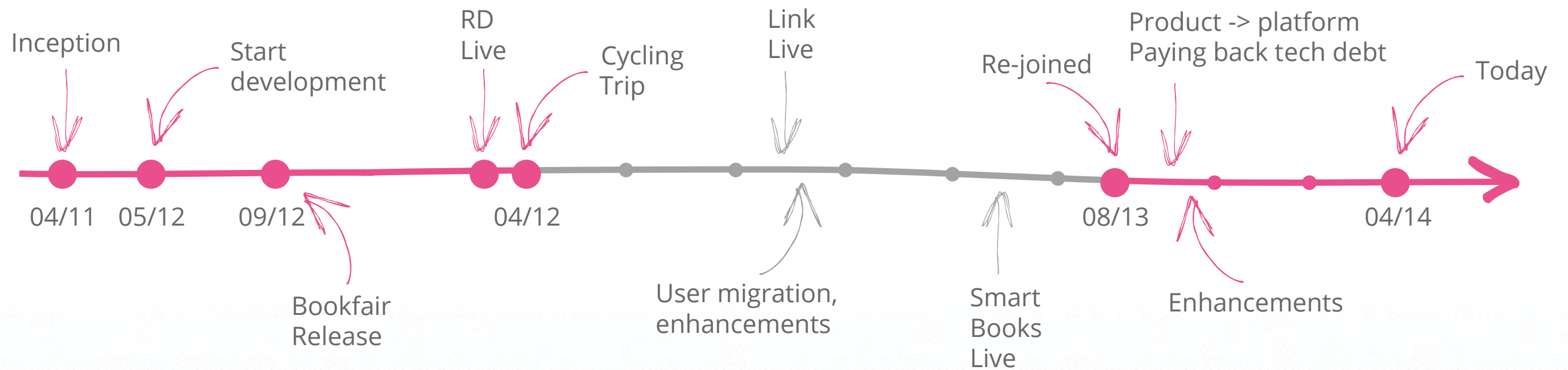
Context



SpringerLink



Timeline



Looking back: why scala?

- Increase productivity
- Be more attractive employer
- Team decision

Looking back: good vs bad

Good

- Functional programming
- Terse syntax
- JVM ecosystem
- Gentle learning curve
- DSL friendly syntax
- Motivated team

Bad

- Tool support
- Compilation times
- Language complexity [#moreRope](#)

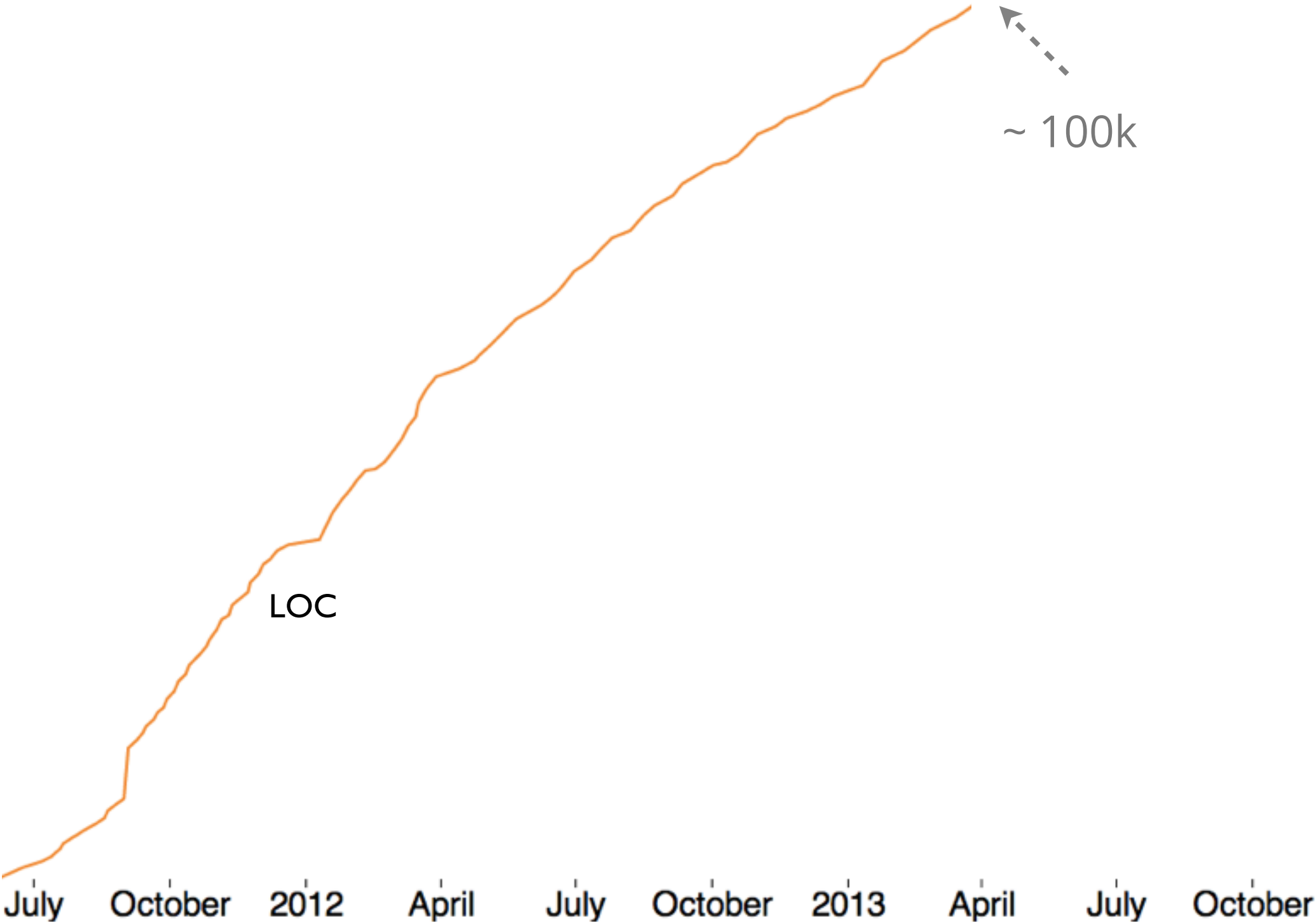
Fast-forward

Fast-forward (Aug 2013)

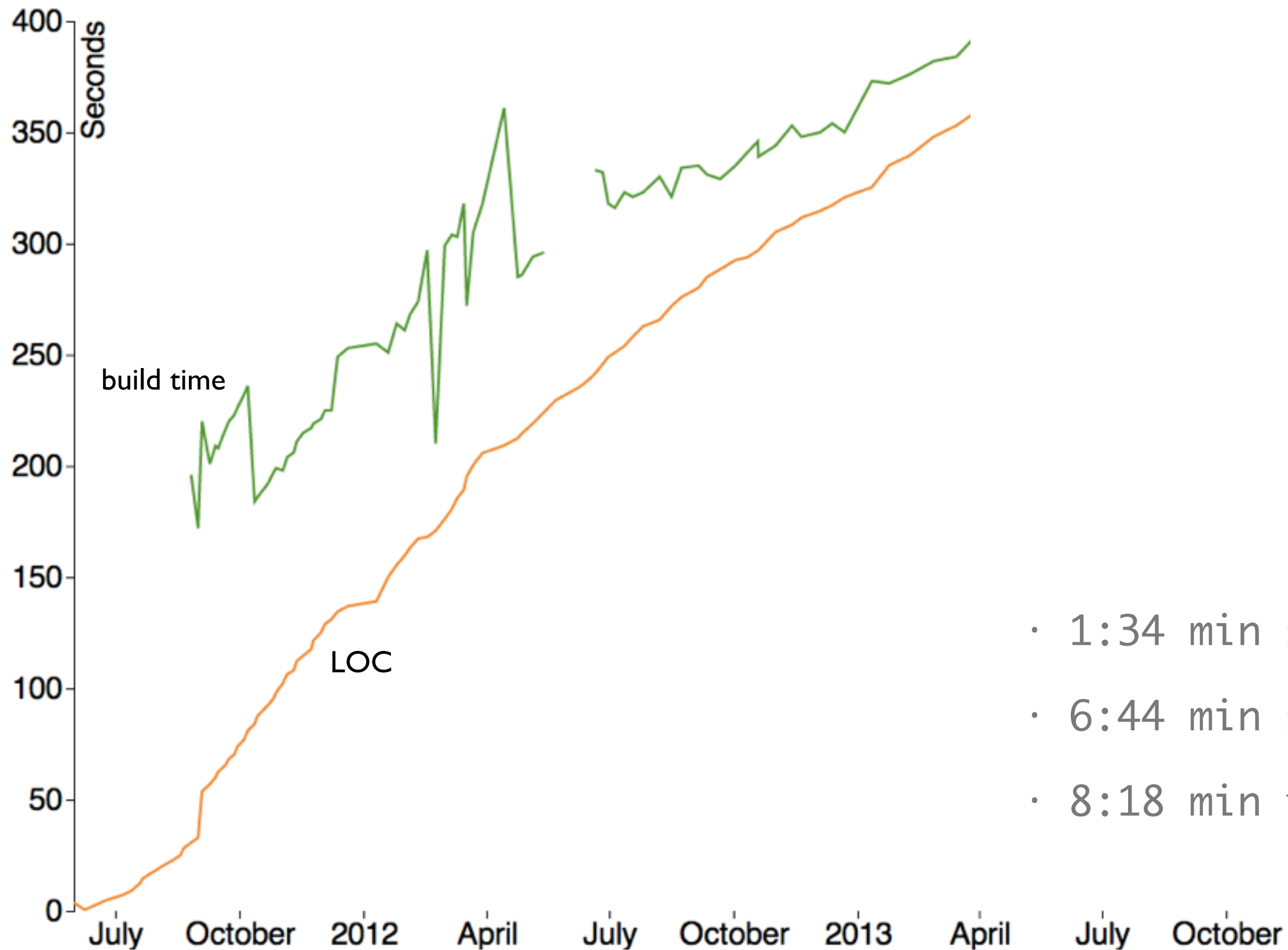
- 2.5 years into project
- 1.5 years of weekly live releases
- 100k LOC
- >10k commits
- >90 committers
- **Poor feedback loops ***
- **Lots of accidental complexity***

* not all related to Scala - to be fair

Trend (2 years)



Trend (2 years)



What did we do?

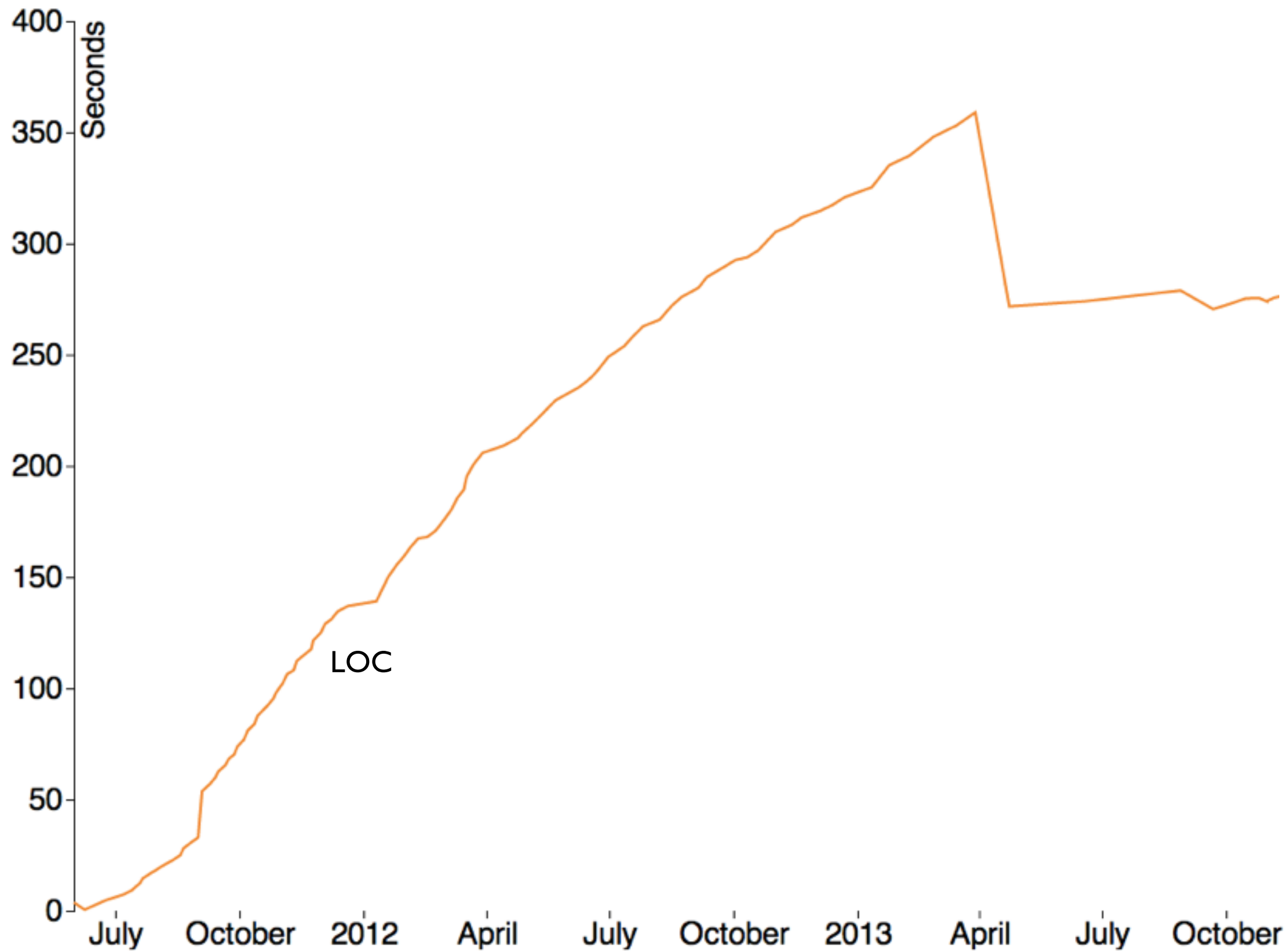
What did we do

- Reduced build time
- Improved feedback loops
- Reduced accidental complexity

Build time

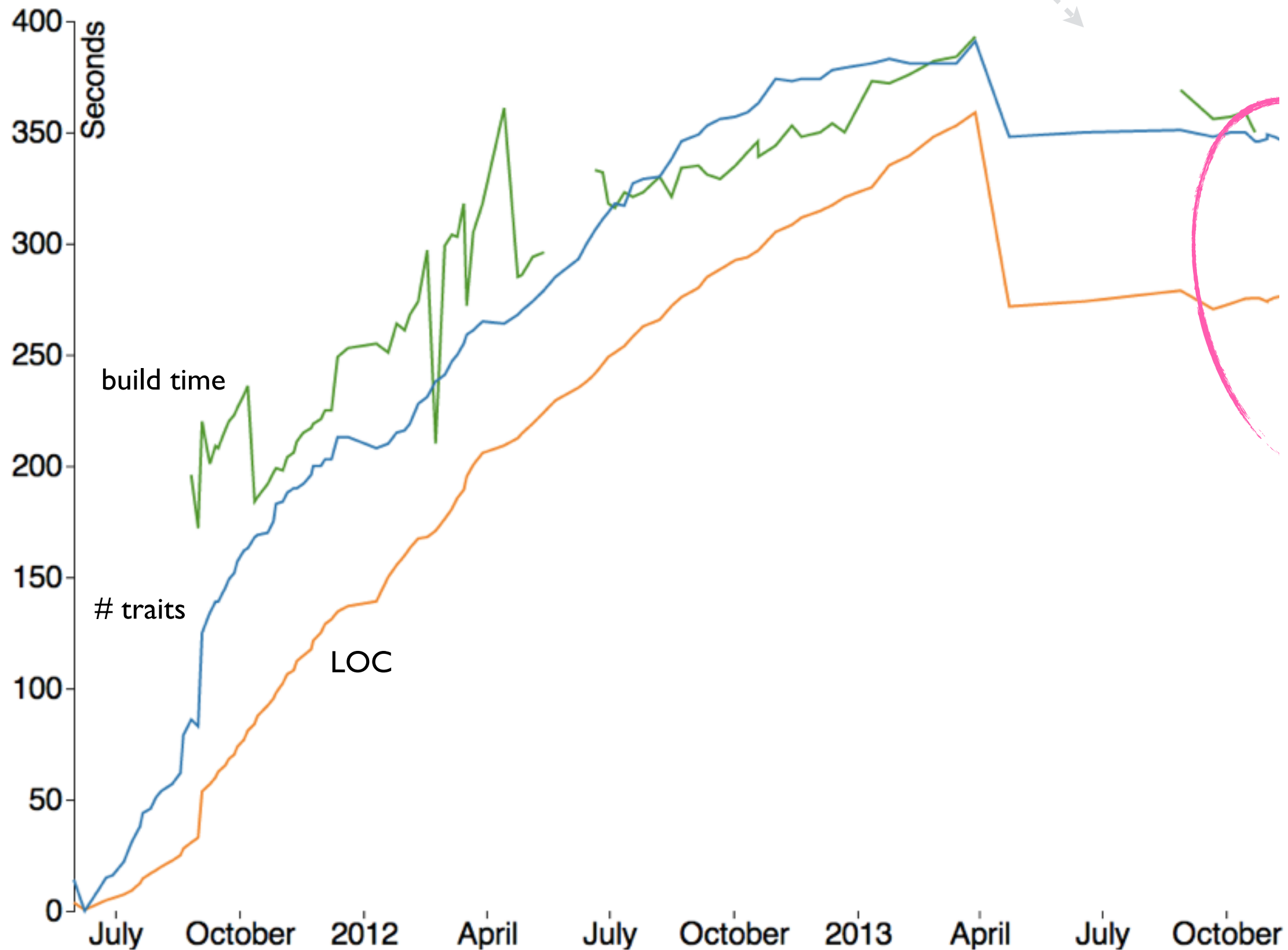
- Reduced size of codebase (broke off vertical slices, pulled out APIs, pulled out libraries, removed unused features, removed low-value tests, etc.)
- Reduced usage of certain language features (esp. traits and implicits)

Trend (Dec 2013)



Trend (Dec 2013)

unable to compile on
13" macbook



The problem with traits

- Will re-compile on every class the trait is mixed in
- Slows down dev-build cycle
- Will result in byte code bloat
- Will compile **a lot** slower

For faster compile times:

- Use pure traits
- Use old-school composition for code re-use
- Use pure functions via imports (e.g. `import Foo._`)
- If unavoidable, use inheritance for code re-use

Build time

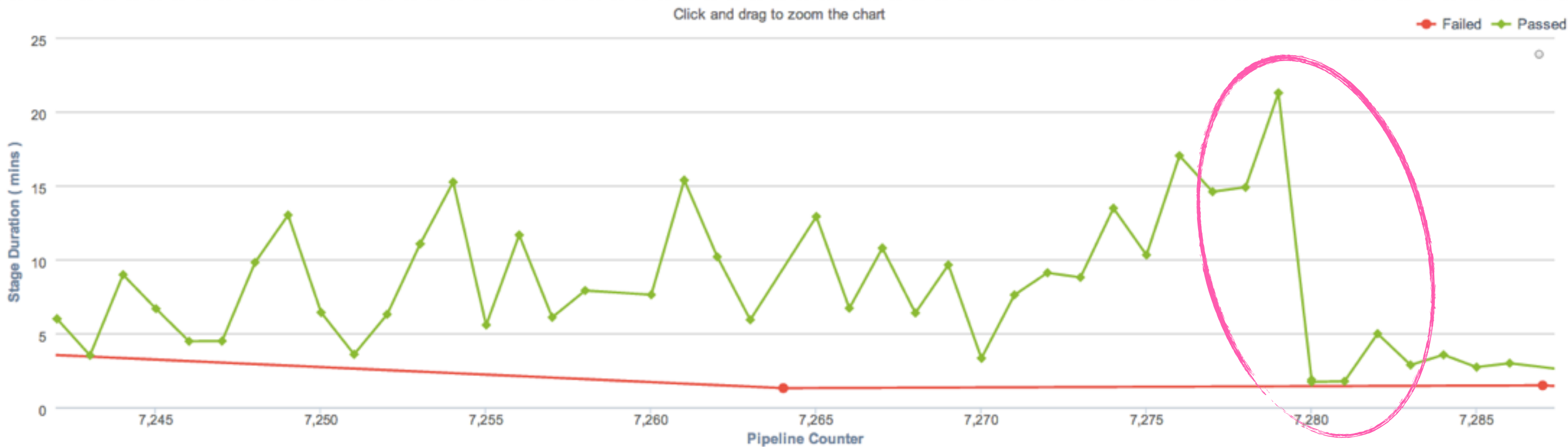
- 1:34 min src/main
- 6:44 min src/test
- 8:18 min total



- 0:24 min src/main
- 3:11 min src/test
- 3:35 min total

Build time (on CI server)

- Incremental compilation on CI
- Only one dedicated CI agent
- Physical build servers
- CPUs with higher clock speed



Complexity

Complexity

- There's still a lot of code in our codebase that is hard to read
- It seems to be very easy to shoot yourself in the foot with Scala
- Scala *is* complex (and that's why scalac will never be as fast as javac)

Invariant/covariant/contravariant types (T , $+T$ and $-T$)

Refined types (`new Foo {...}`)

Structural types (`x: {def y: Int}`)

Path dependant types (`a.B`)

Specialized types (`@specialized`)

Self types (`this =>`)

Projection types (`A#B`)

Existential types (`M[_]`)

Type bounds ($<:$, $>:$)

Type constraints ($==:$, $<:<$ and $<%\<$)

Type members (`type T`)

Type aliases (`type T = Int`)

Type classes (`(implicit ...)`)

View bounds ($<%$)

Higher kinded types ($* \Rightarrow *$)

F-Bounded type polymorphism (`M[T <: M[T]]`)

Not opinionated

- Many ways to do the same thing
- Coding conventions help, but only so much*

```
def foo() = "foo"  
def bar = "bar"
```

```
foo  
foo()  
bar  
bar()           // won't compile
```

```
def baz(x: String) = x  
"x".charAt(0)  
"x" charAt(0)    // won't compile  
"x".charAt 0     // won't compile  
"x" charAt 0  
baz("x")  
baz "x"          // won't compile
```

```
list.foreach { x => println(x) }  
list.foreach ( x => println(x) )  
list.foreach { println(_) }  
list.foreach ( println(_) )  
list foreach { x => println(x) }  
list foreach ( x => println(x) )  
list foreach { println(_) }  
list foreach ( println(_) )
```

```
if (foo) "x" else "y"
```

```
foo match {  
  case true => "x"  
  case _ => "y"  
}
```

* For example: <http://twitter.github.io/effectivescala/>

Surprises

```
List(1, 2, 3).toSet
```

```
scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

```
List(1, 2, 3).toSet()
```

```
Boolean = false
```


Implicits

- Can make it very hard to read code
- Tool support is very bad
- Impacts compilation time
- Surprising behaviour (esp. when used with overloaded methods or optional params)

Tooling

- Tool support is still very basic
- Makes it hard to continuously refactor (which means people are less likely to do it)

```
def handle(response: HttpResponse, request: HttpRequest)
```



no luck with “change signature”
refactoring support

Trait entanglements

- Makes it difficult to reason about behaviour

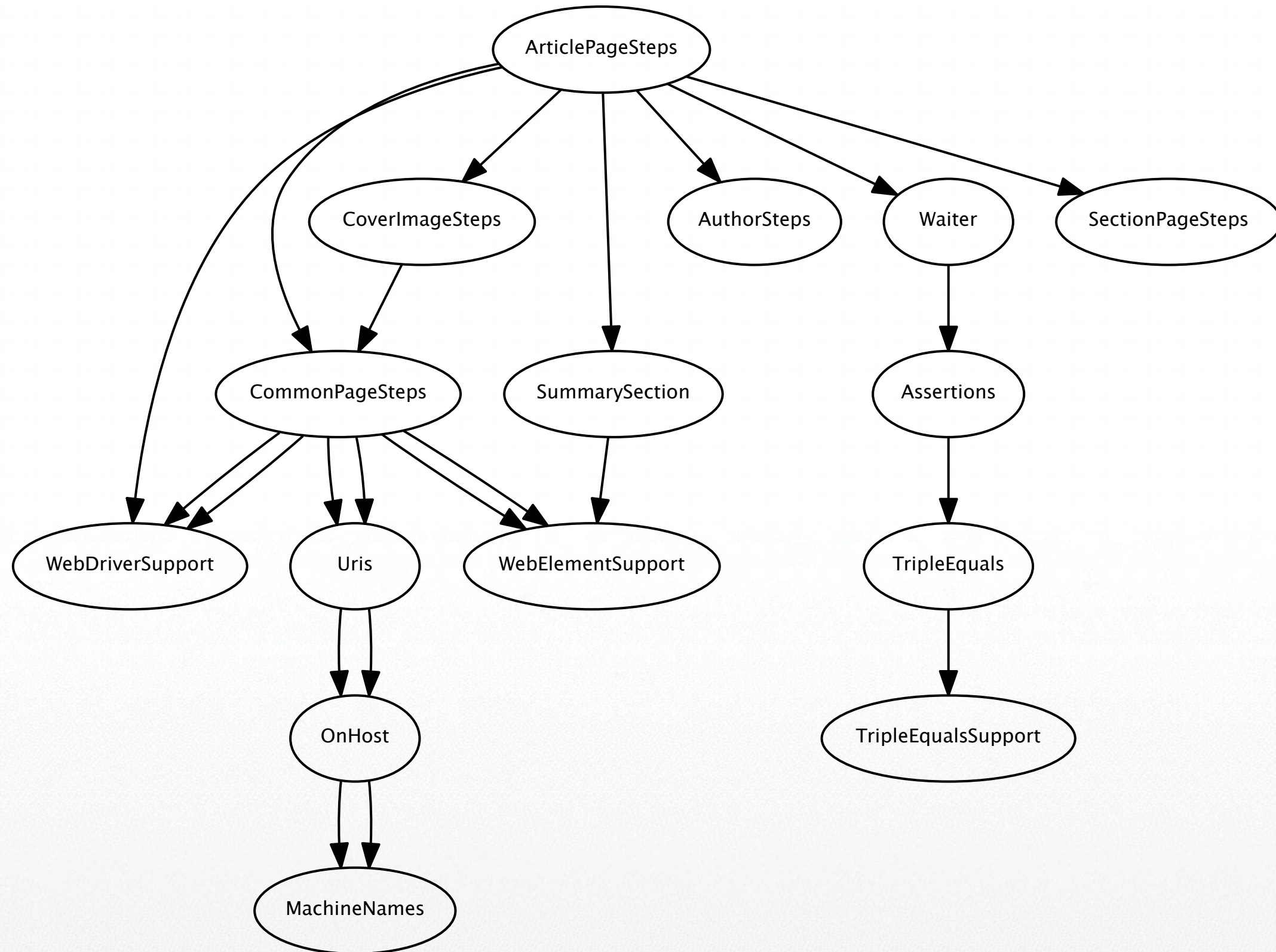
```
trait A {  
  def foo = "a"  
}
```

```
trait B extends A {  
  override def foo = "b"  
}
```

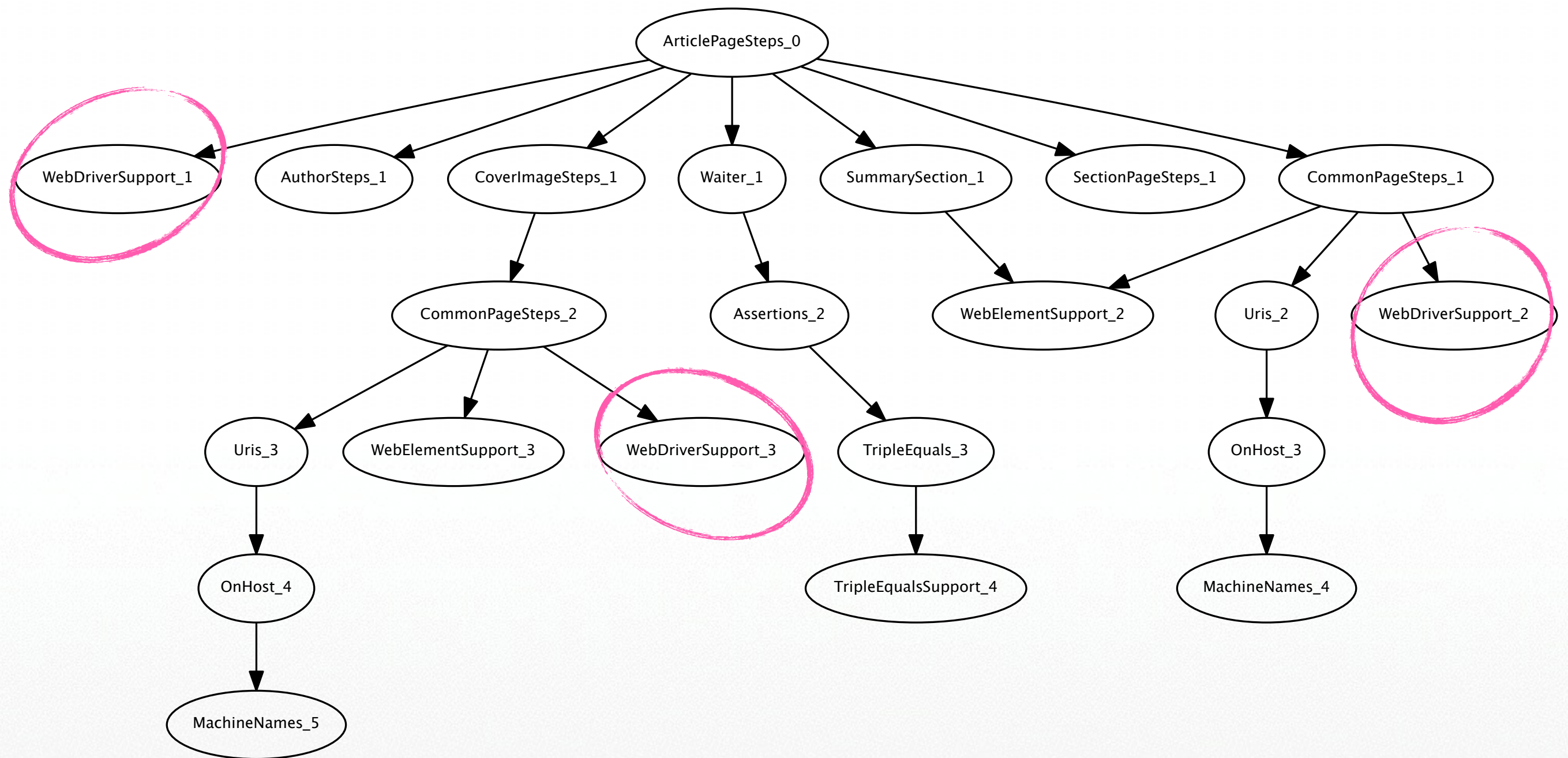
```
class C extends A with B  
new C().foo  
"b"
```

```
class D extends B with A  
new D().foo  
"b"
```

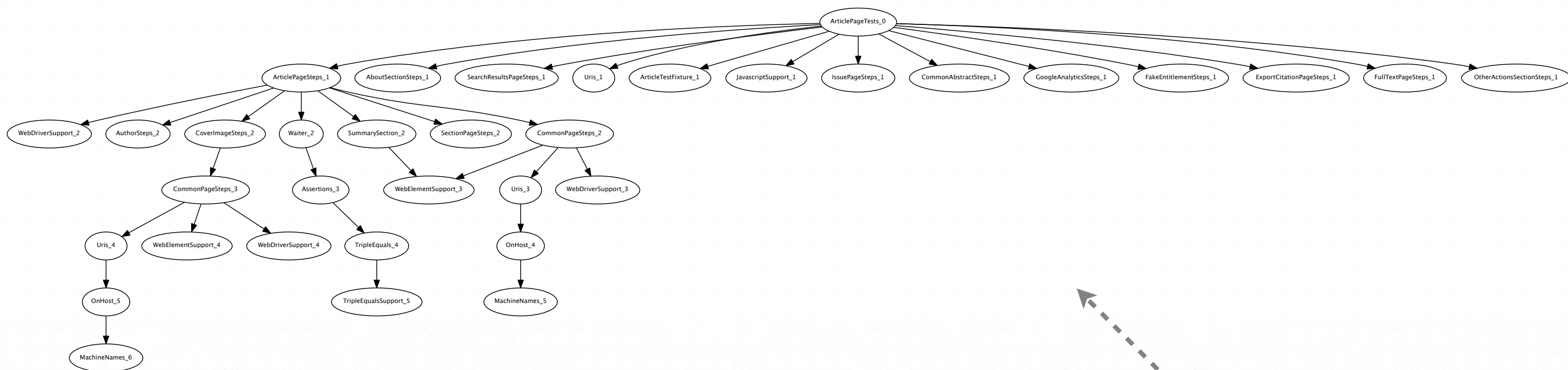

Trait entanglements (2)



Trait entanglements (3)



Trait entanglements (4)



Imagine many more circle here

So, what's next?

Today

- We've delivered successfully using Scala
- Don't think we're more productive (pure gut feeling, though)
- We try to stick to the good parts (conventions, functional programming, pattern matching, etc.)
- Complexity, slow compilation and lack of tool support are real problems

The future

- No urgency to move away from Scala or re-write existing systems
- Java 8 is an alternative
- Smaller teams and apps will probably lead to more polyglotism (and less Scala)

Thanks
!!
😊

<http://joinit.springer.com>

@patforna

patric.fornasier@springer.com