How does incremental compilation work in Scala 3?

Jamie Thompson

scalacenter
For open source. For education.

- Twitter: @bishabosha
- Email: scala.center@epfl.ch

Agenda

Explaining the Scala Build

What happens when you build a project?

How do build tools optimise?

Incremental compilation, **Pipelined** builds

Takeaways

Which steps can you take today to improve build times?

What does the future hold?

Can we add more innovations to speed up builds?

Research

EPFL

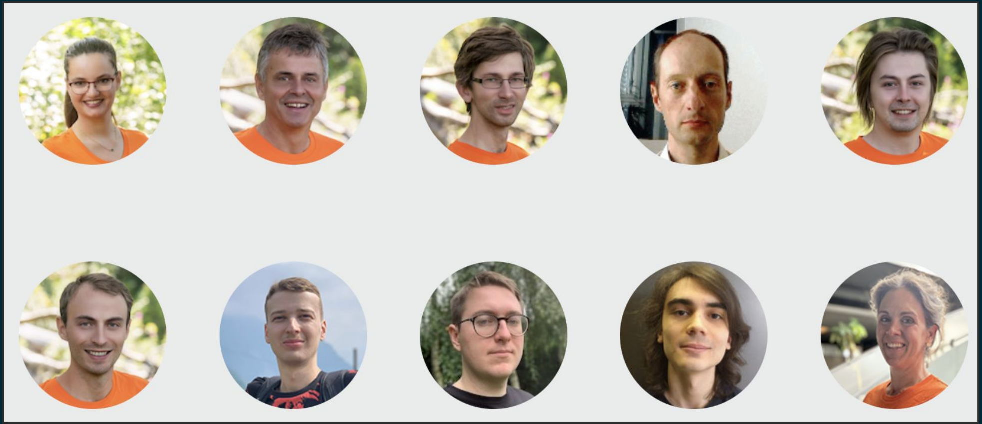
scalacenter
For open source. For education.

Open Source

8000+ Github repos
1,200,000+ artifacts

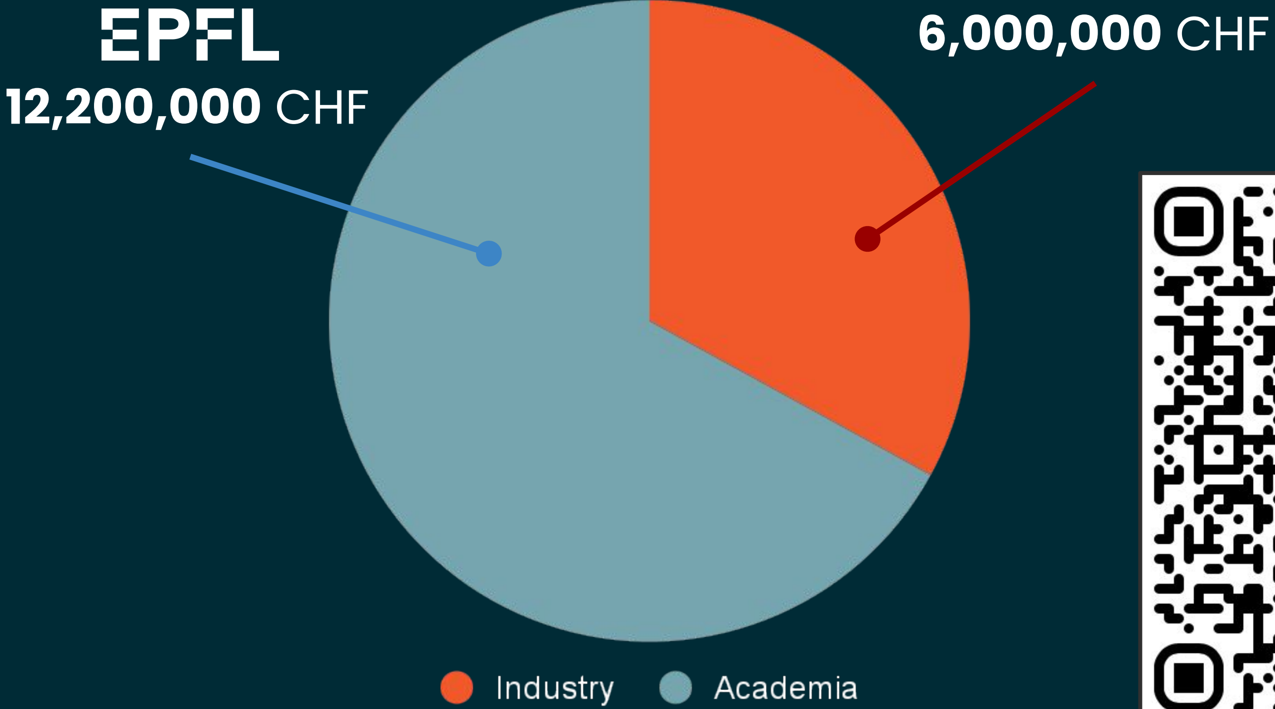
Industry

scalacenter
For open source. For education.



- Created in 2016 at EPFL
- Not for profit
- Team of 10 people:
  - administration
  - communication
  - engineering
- Advisory Board:
  - 2 Community Representatives
  - 5 Companies

Last 10 years of Scala 3 investment

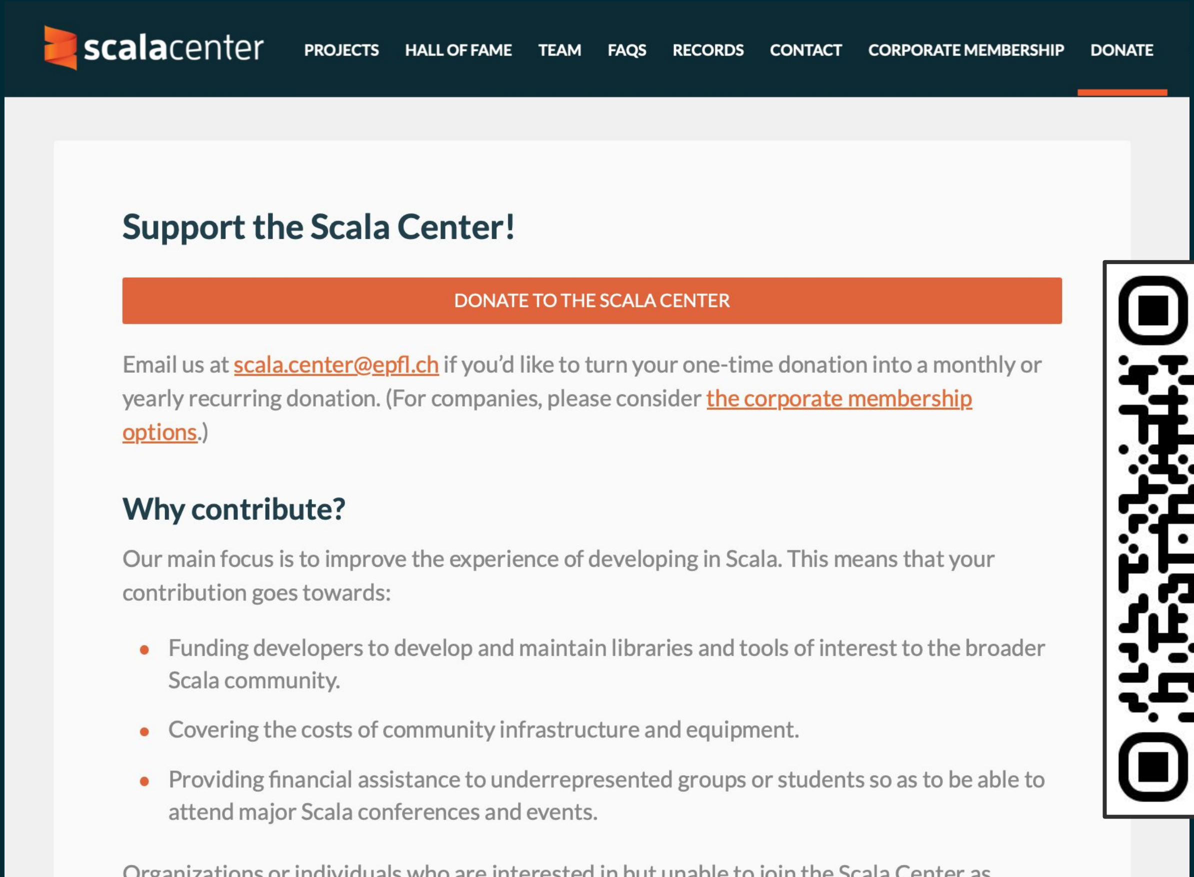


scalacenter
For open source. For education.

VIRTUSLAB

Lightbend

Fundraising Campaign



scalacenter PROJECTS HALL OF FAME TEAM FAQS RECORDS CONTACT CORPORATE MEMBERSHIP DONATE

Support the Scala Center!

DONATE TO THE SCALA CENTER

Email us at scala.center@epfl.ch if you'd like to turn your one-time donation into a monthly or yearly recurring donation. (For companies, please consider the corporate membership options.)

Why contribute?

Our main focus is to improve the experience of developing in Scala. This means that your contribution goes towards:

- Funding developers to develop and maintain libraries and tools of interest to the broader Scala community.
- Covering the costs of community infrastructure and equipment.
- Providing financial assistance to underrepresented groups or students so as to be able to attend major Scala conferences and events.

Organizations or individuals who are interested in but unable to join the Scala Center as

Explaining the Scala Build

“Premature optimisation is the root of all evil” - *Donald Knuth*

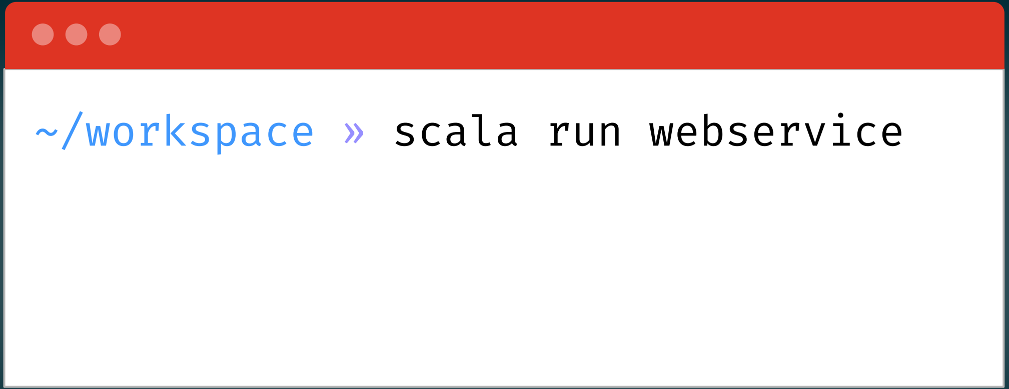
Example: Small Server App

1 project directory

50 source files

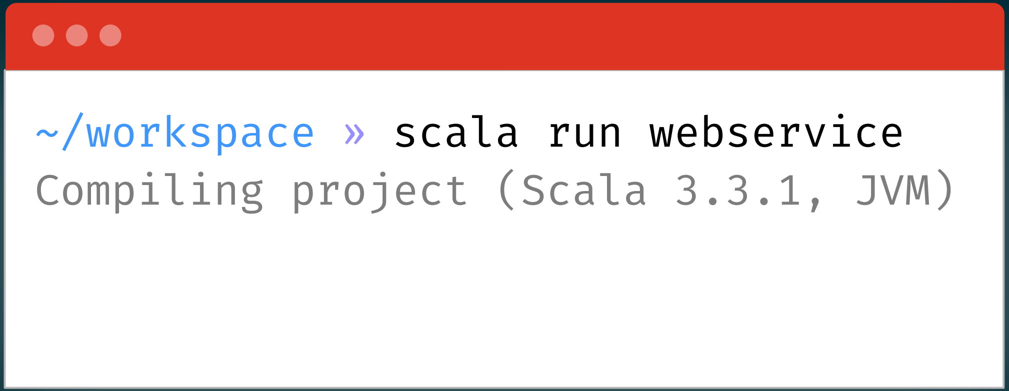
10 library dependencies

Example: Small Server App



```
~/workspace » scala run webservice
```

Example: Small Server App

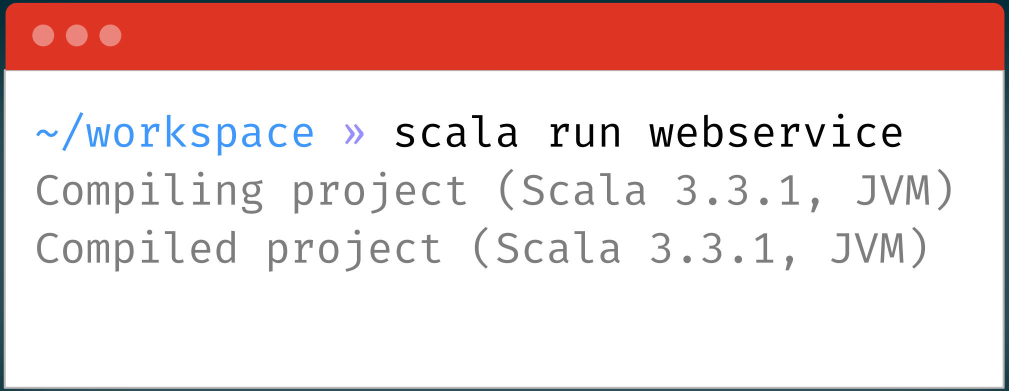


```
~/workspace » scala run webservice
Compiling project (Scala 3.3.1, JVM)
```

1. **Fetch** dependencies
2. **Generate** source files

once per config update

Example: Small Server App



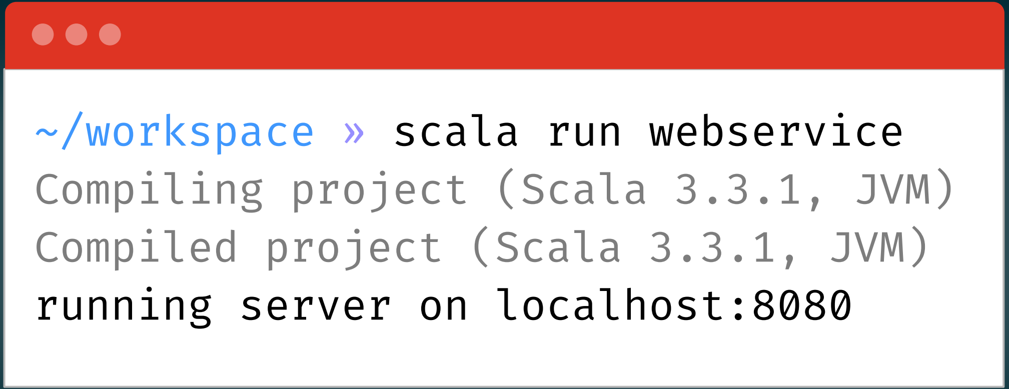
```
~/workspace » scala run webservice
Compiling project (Scala 3.3.1, JVM)
Compiled project (Scala 3.3.1, JVM)
```

1. **Fetch** dependencies
2. **Generate** source files
3. **Compile** source files to runtime platform

once per config update

slow!!!

Example: Small Server App



```
~/workspace » scala run webservice
Compiling project (Scala 3.3.1, JVM)
Compiled project (Scala 3.3.1, JVM)
running server on localhost:8080
```

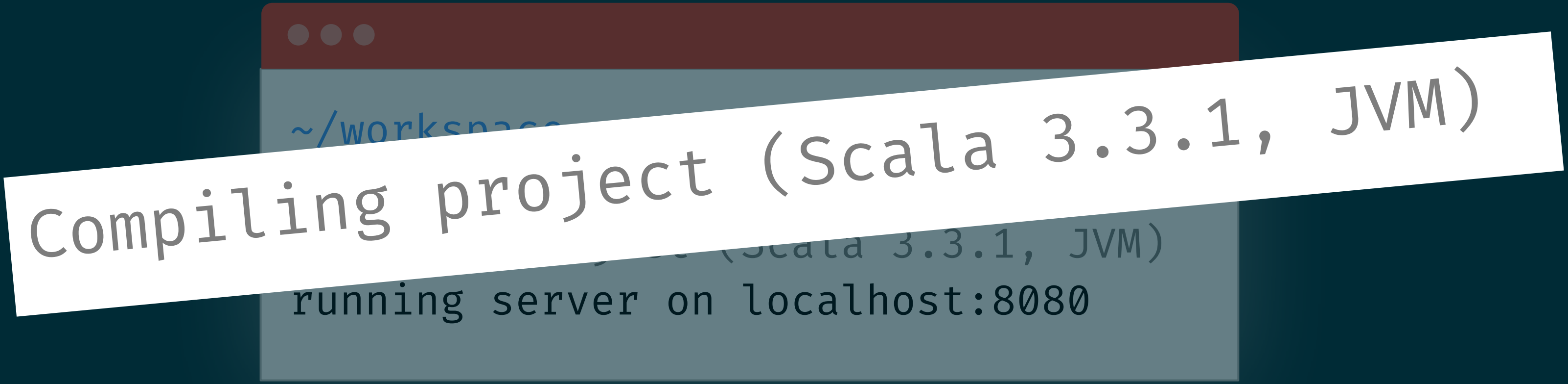
1. **Fetch** dependencies
2. **Generate** source files
3. **Compile** source files to runtime platform
4. **Create** a launcher and execute it

$\left.\begin{matrix}1.\\2.\end{matrix}\right\}$ *once per config update*

3. *slow!!!*

4. *fast*

Example: Small Server App



```
~/workspac
          ject (Scala 3.3.1, JVM)
running server on localhost:8080
```

```
Compiling project (Scala 3.3.1, JVM)
```

1. **Fetch** dependencies
2. **Generate** source files

once per config update

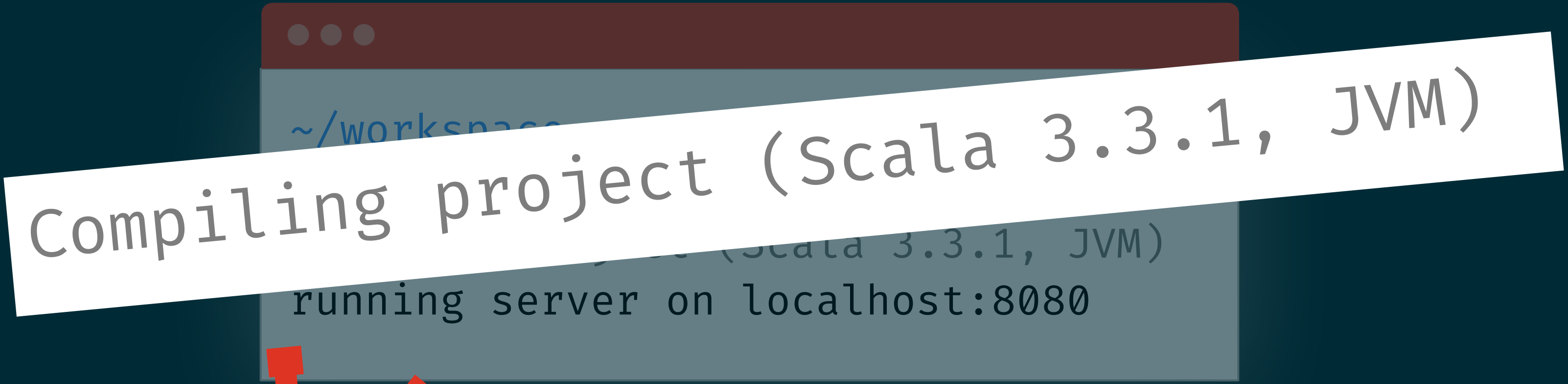
3. **Compile** source files to runtime platform

slow!!!

4. **Create** a launcher and execute it

fast

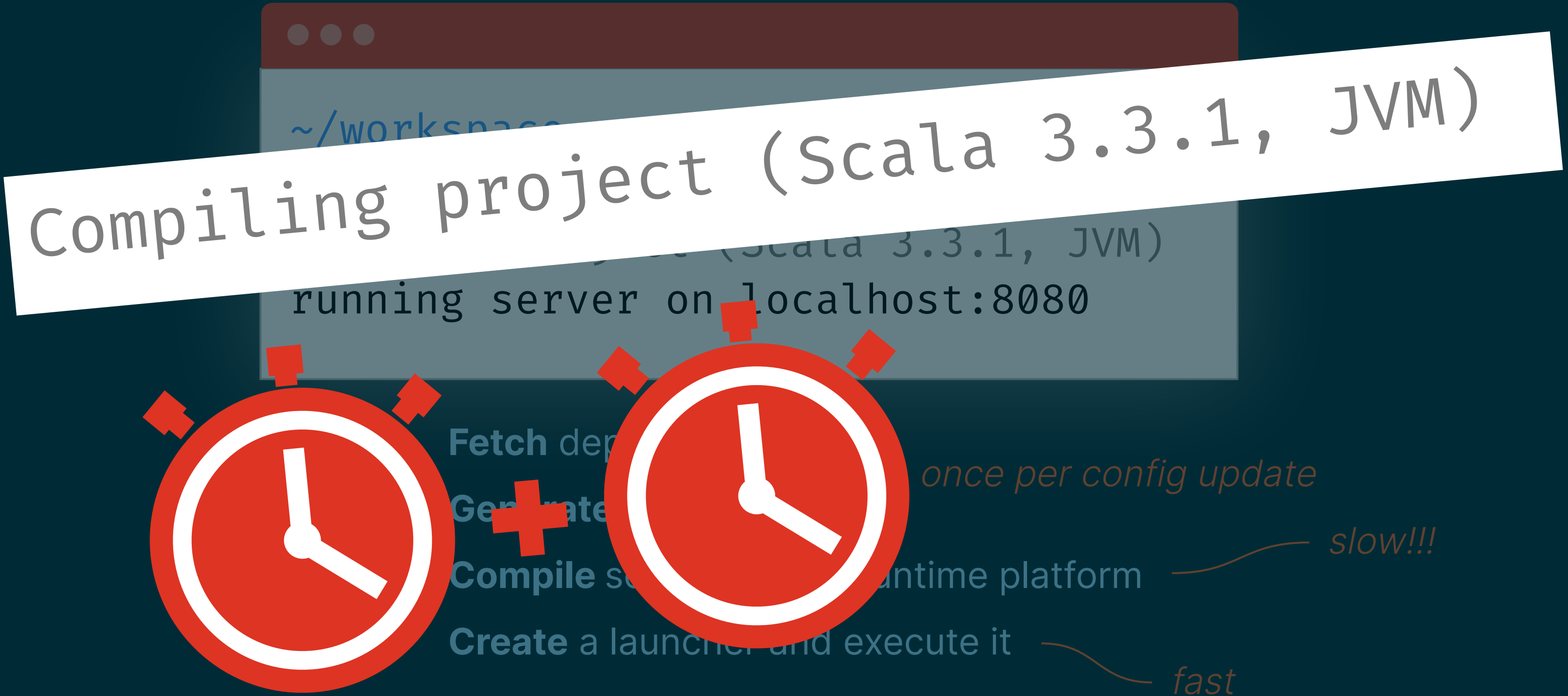
Example: Small Server App



```
~/worksp
Compiling project (Scala 3.3.1, JVM)
running server on localhost:8080
```

Fetch dependencies
Generate source files — *once per config update*
Compile source files to runtime platform — *slow!!!*
Create a launcher and execute it — *fast*

Example: Small Server App



```
~/workspace
Compiling project (Scala 3.3.1, JVM)
running server on localhost:8080
```

Compiling project (Scala 3.3.1, JVM)

Fetch dep

Generate

Compile s… untime platform

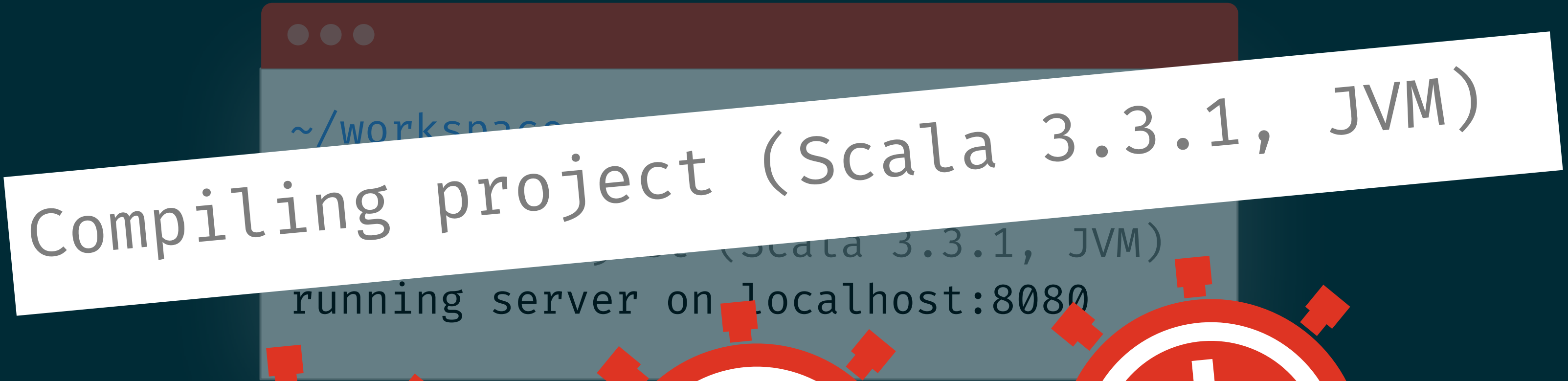
Create a launcher and execute it

once per config update

slow!!!

fast

Example: Small Server App



```
~/worksp
                  (Scala 3.3.1, JVM)
running server on localhost:8080
```

```
Compiling project (Scala 3.3.1, JVM)
```

Fetch dep

Ge   **ate**

o   pe

Compile s   ntime platform

slow!!!

Create a launch   nd execute it

fast

Example: Small Server App

```
~/workspace
                                              JVM)
running ser   r on lo              080
```

Compiling project (Scala 3.3.1, JVM)

* goal *

1. Fetch de end
2. Generate so
3. Compile files runtime platform
4. Creat ncher and execute it

once per config update

slow!!!

fast

Incremental Compilation

Compile **only the changed definitions** and their **uses**, compared to the **previous compilation**.

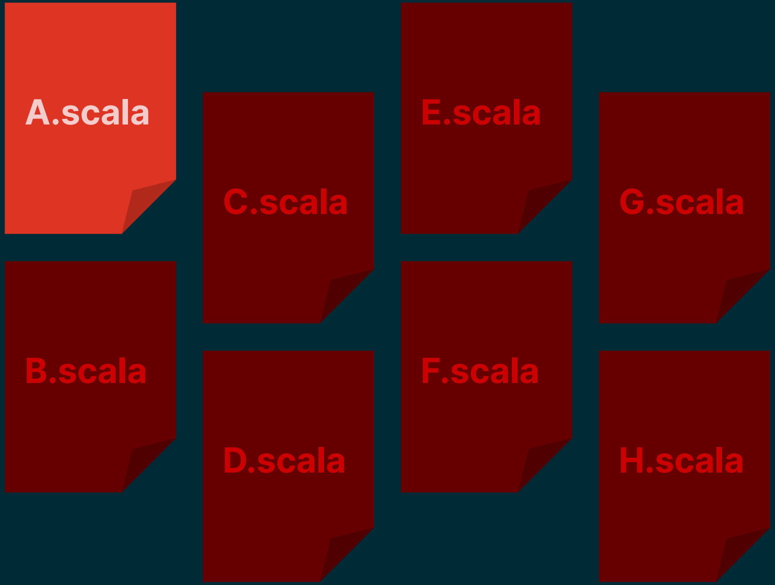
Incremental Compilation

Instead of compiling all of these files...

A.scala
B.scala
C.scala
D.scala
E.scala
F.scala
G.scala
H.scala

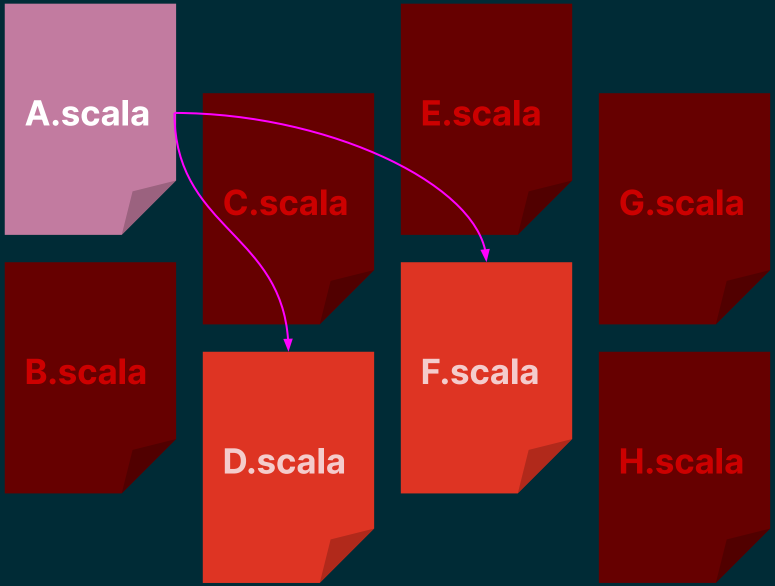
Incremental Compilation

*First, detect **A.scala** has **changed**, compile it.*



Incremental Compilation

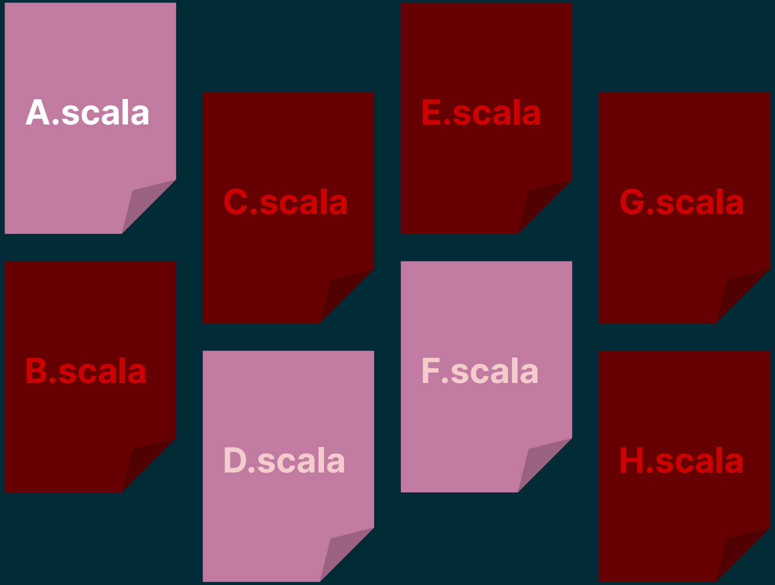
Next, compile the ***dependencies*** *of any* ***changed definitions***



Incremental Compilation

***No more changes**,*
*compilation **success***

✅



Incremental Compilation

The two unbreakable rules

Correct

compile everything that is necessary

Performant

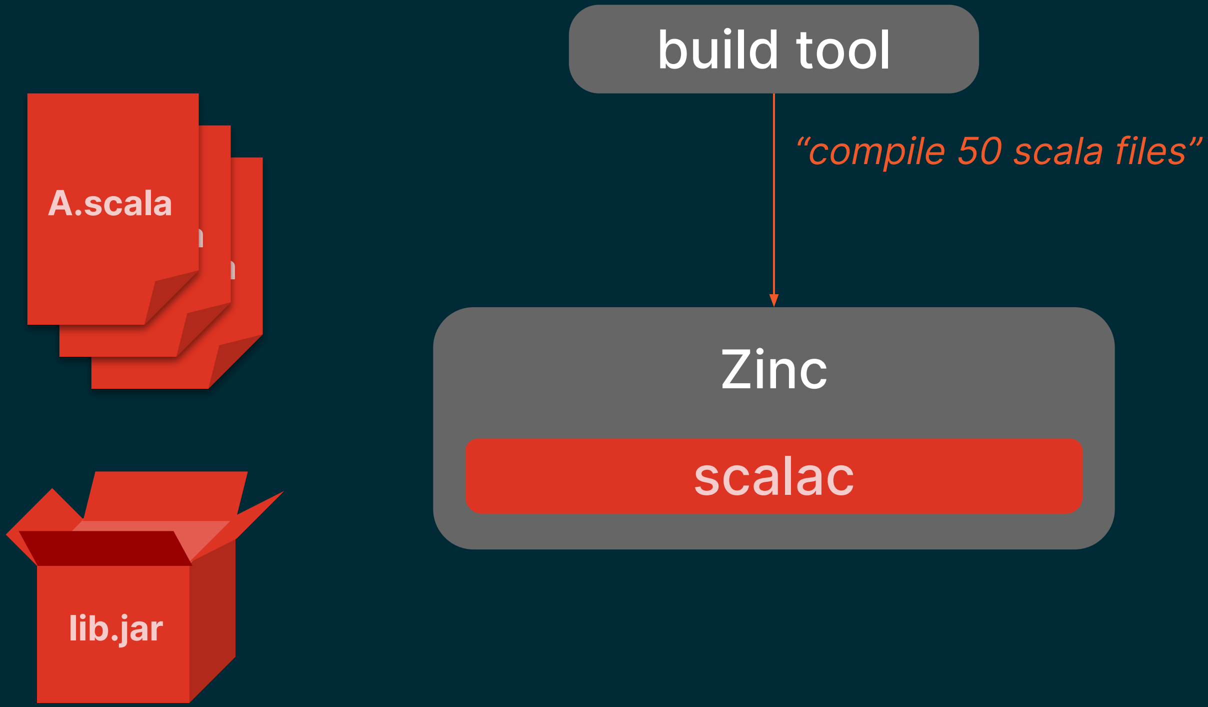
efficient invalidation

don’t compile more than necessary

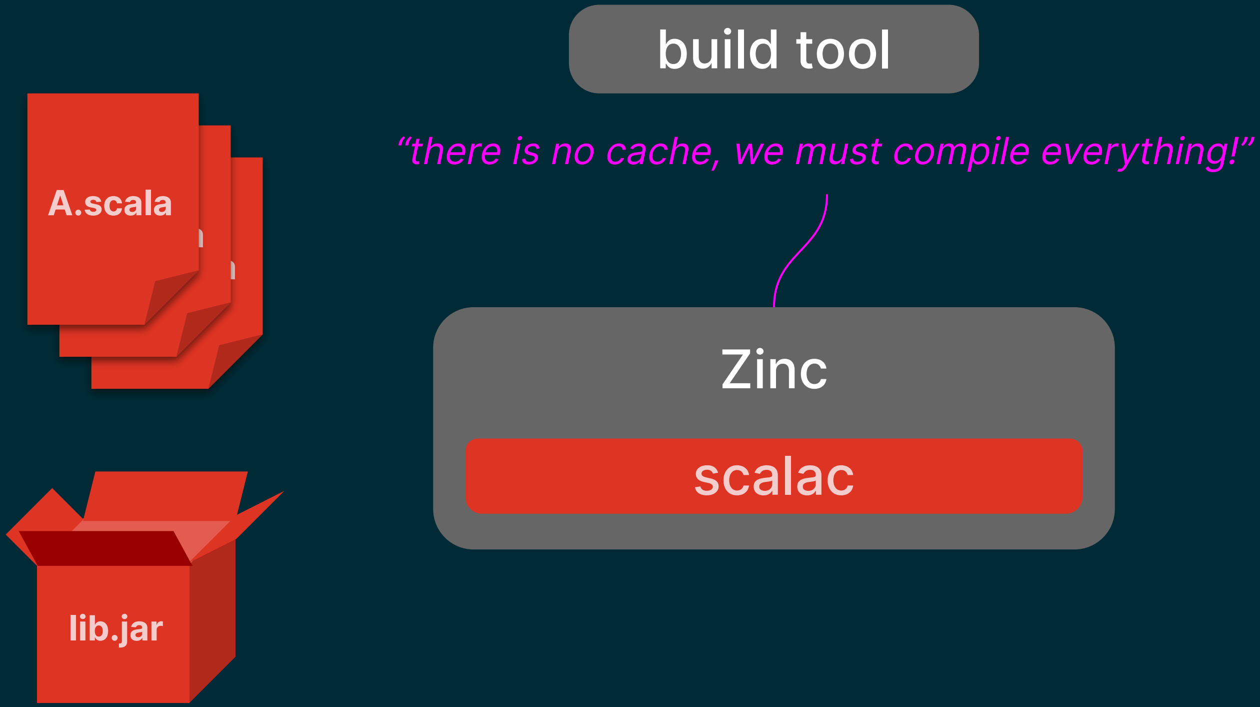
Introducing Zinc

Zinc is an **incremental compiler** for the Scala language, it maximizes **correctness** and **performance** with the **name hashing** algorithm.

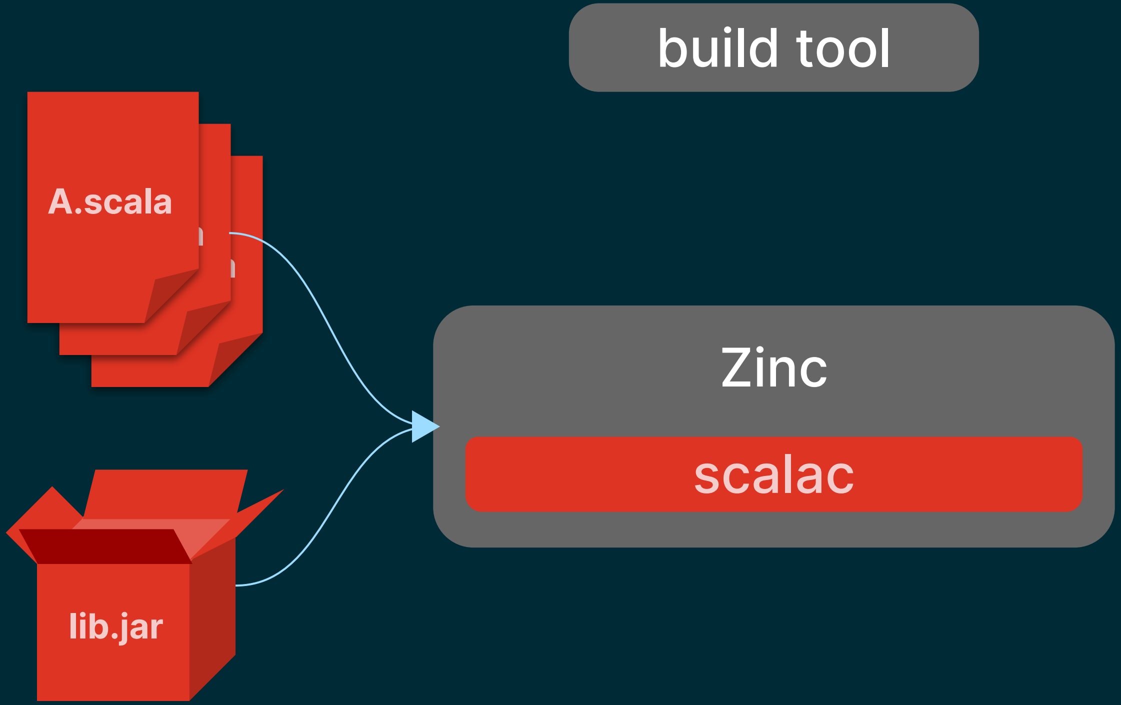
Incremental Compilation



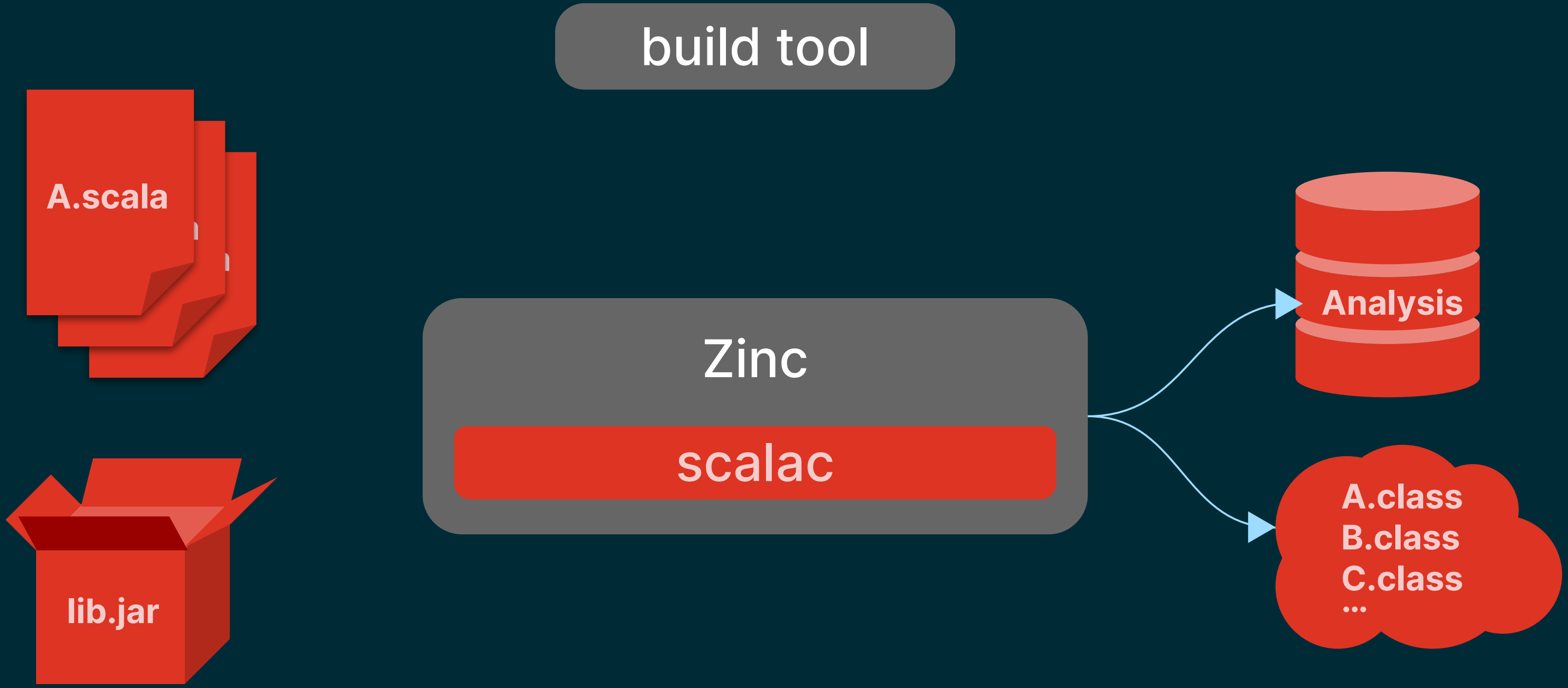
Incremental Compilation



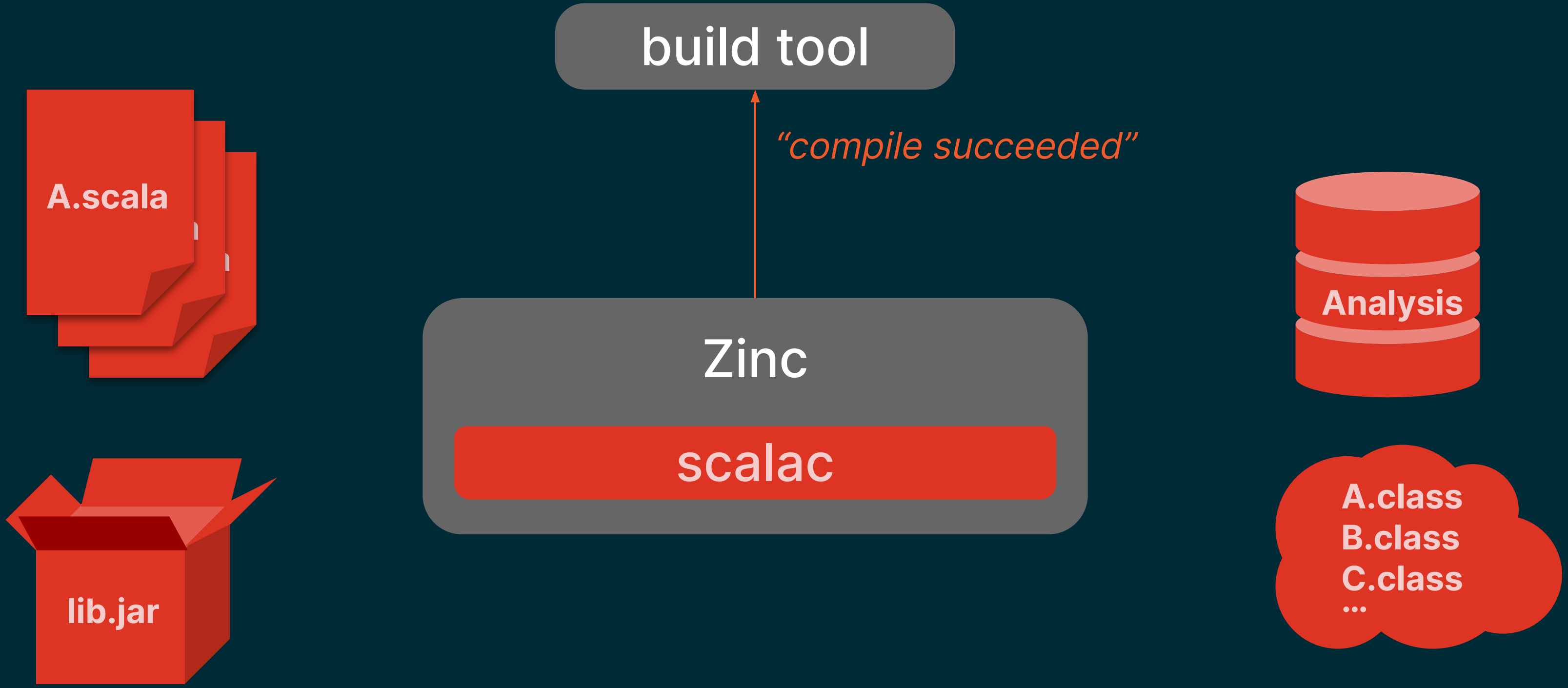
Incremental Compilation



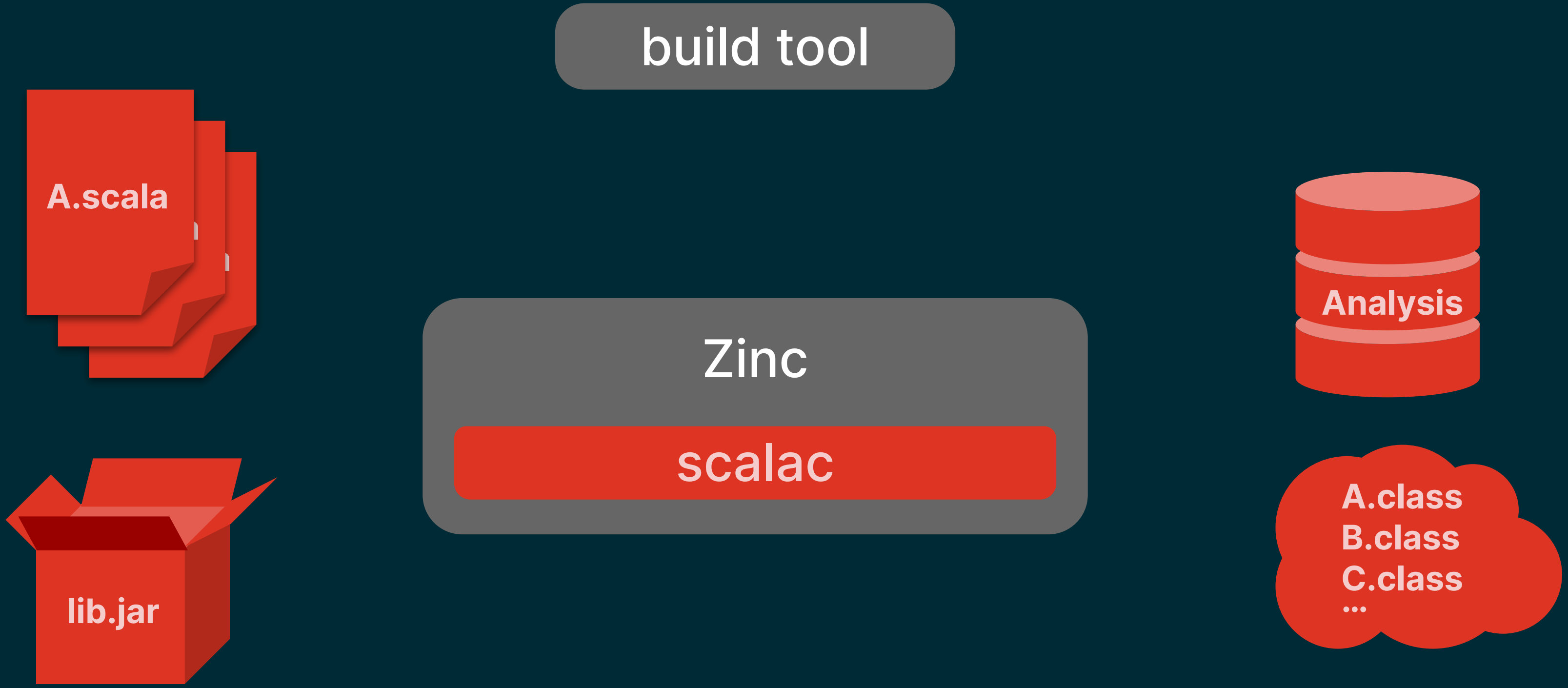
Incremental Compilation



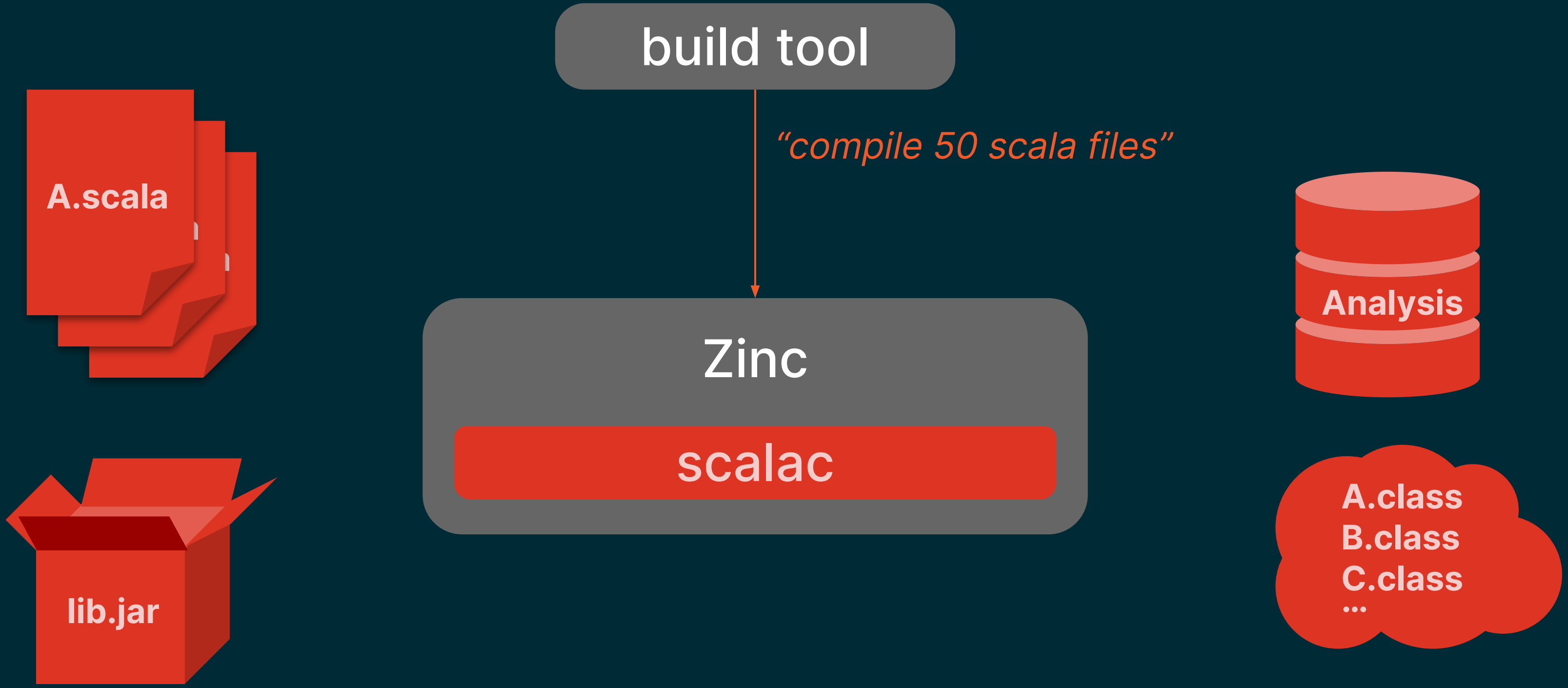
Incremental Compilation



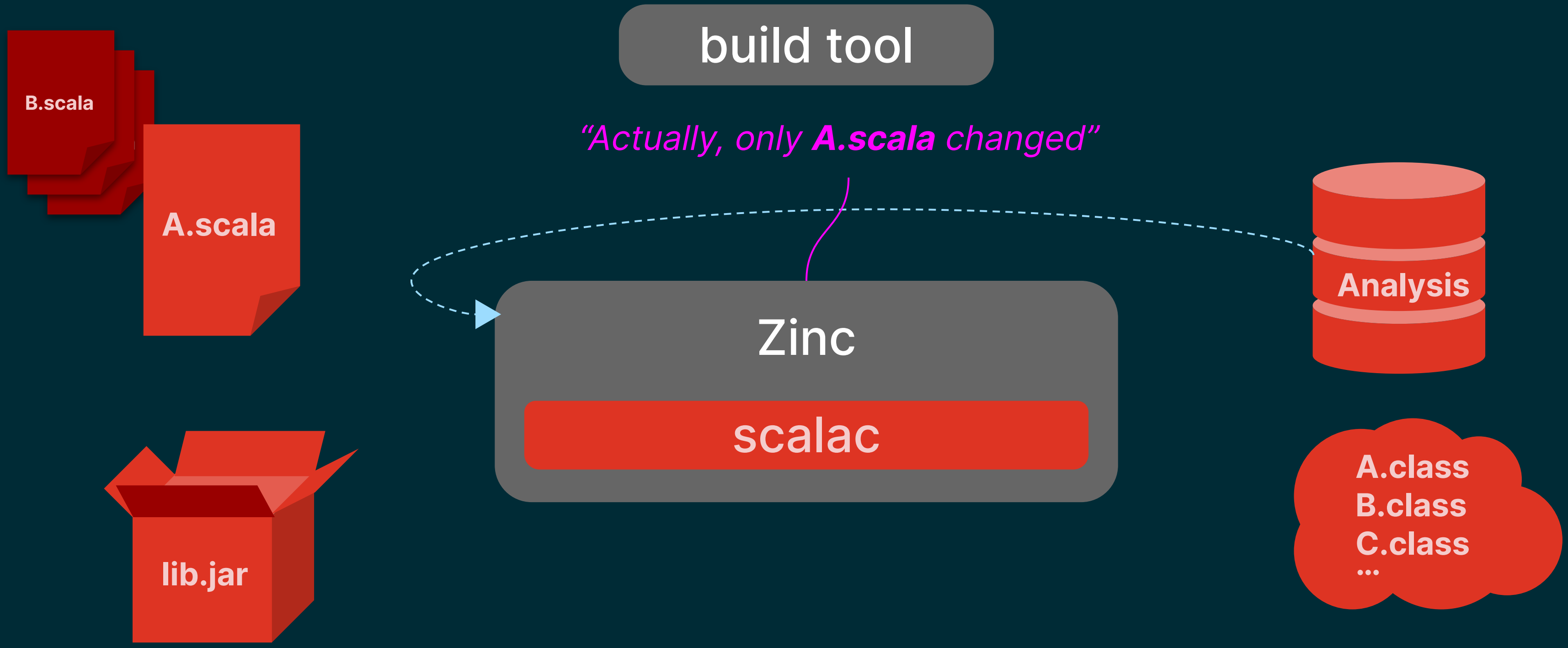
Incremental Compilation



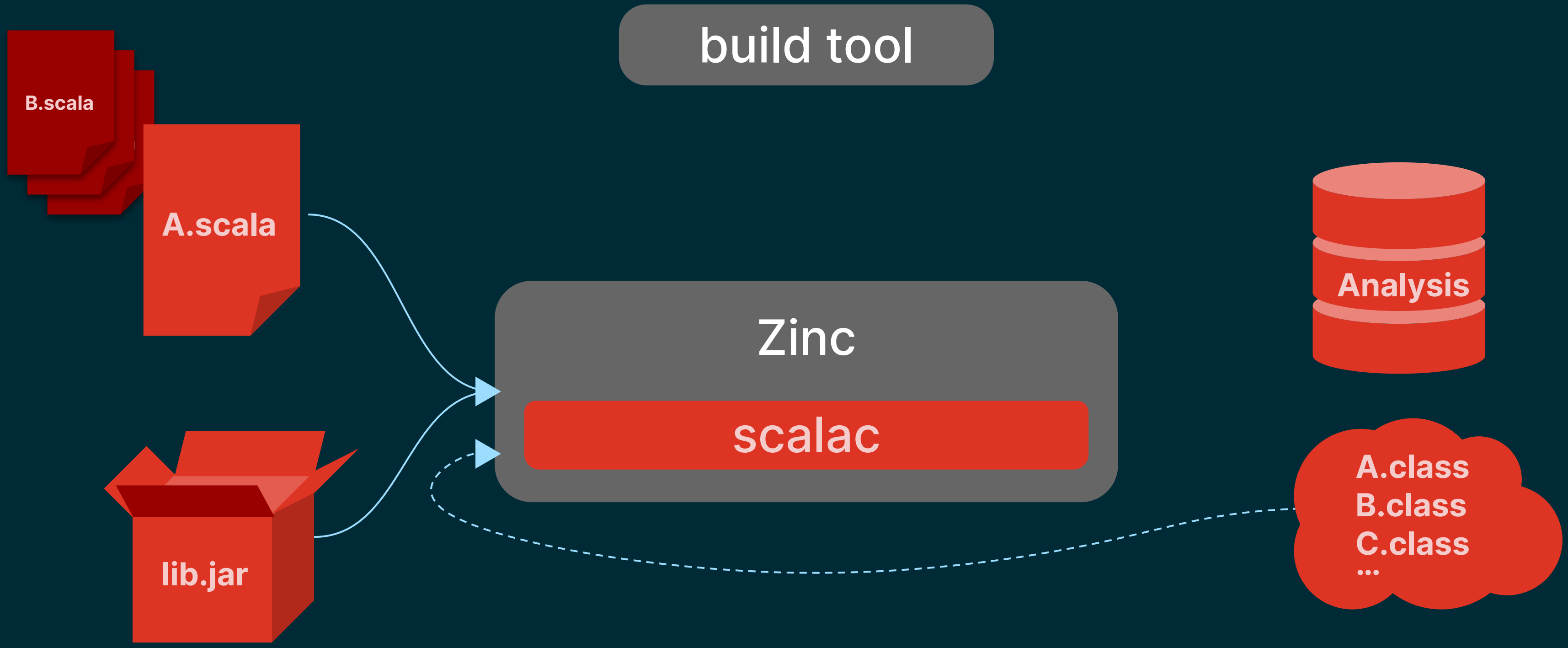
Incremental Compilation



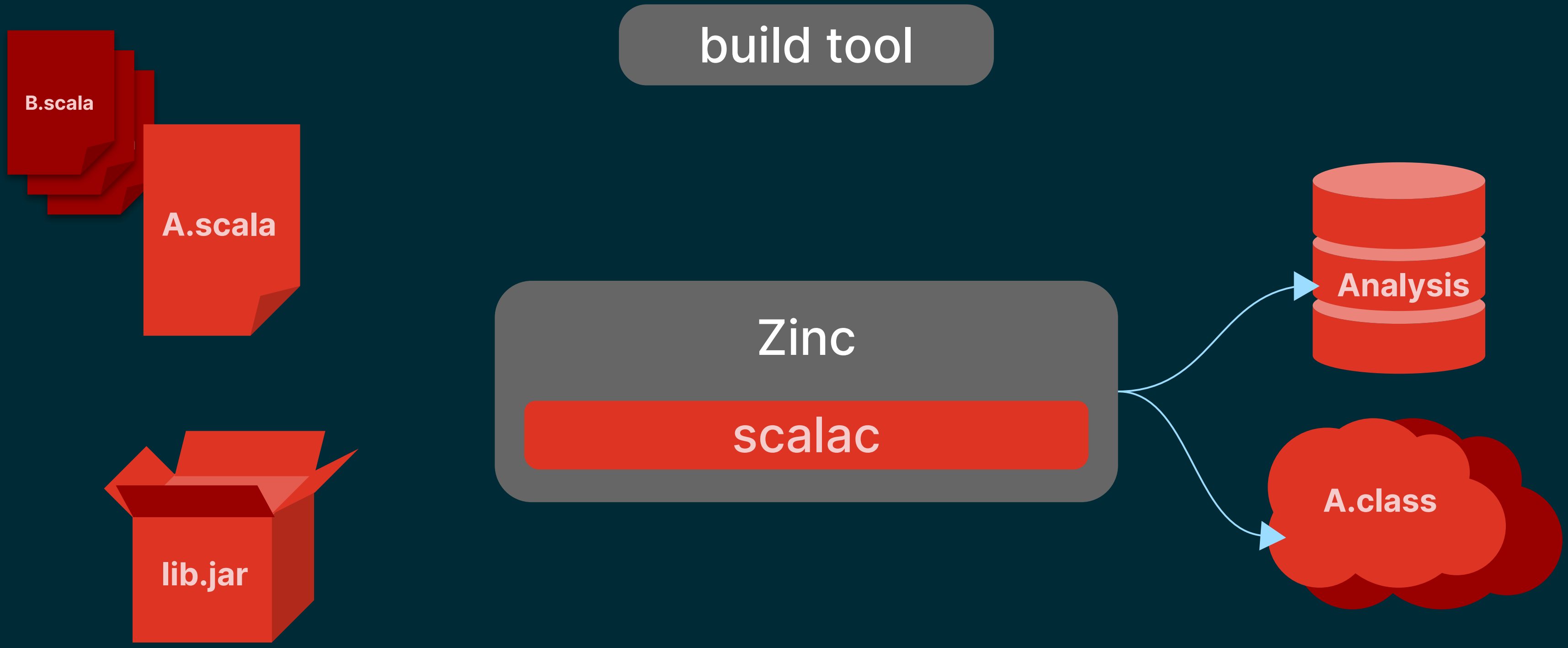
Incremental Compilation



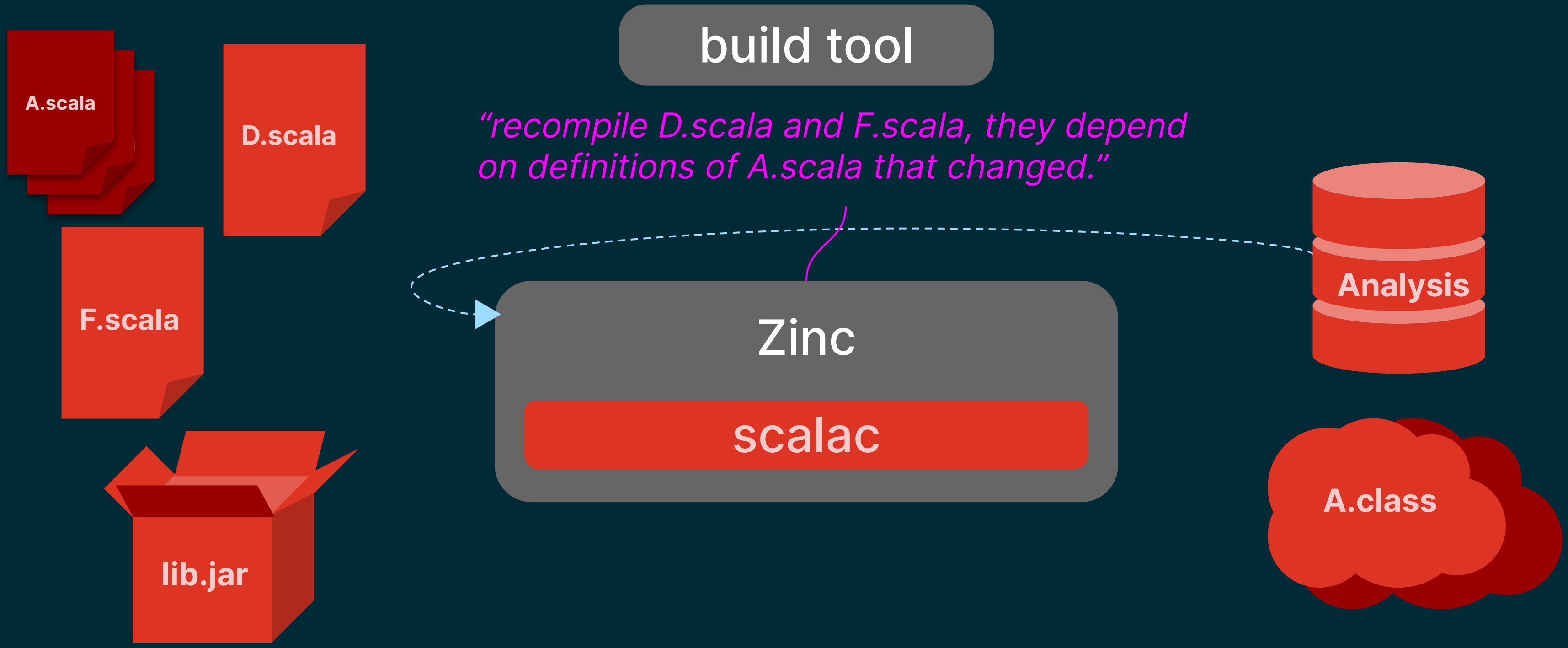
Incremental Compilation



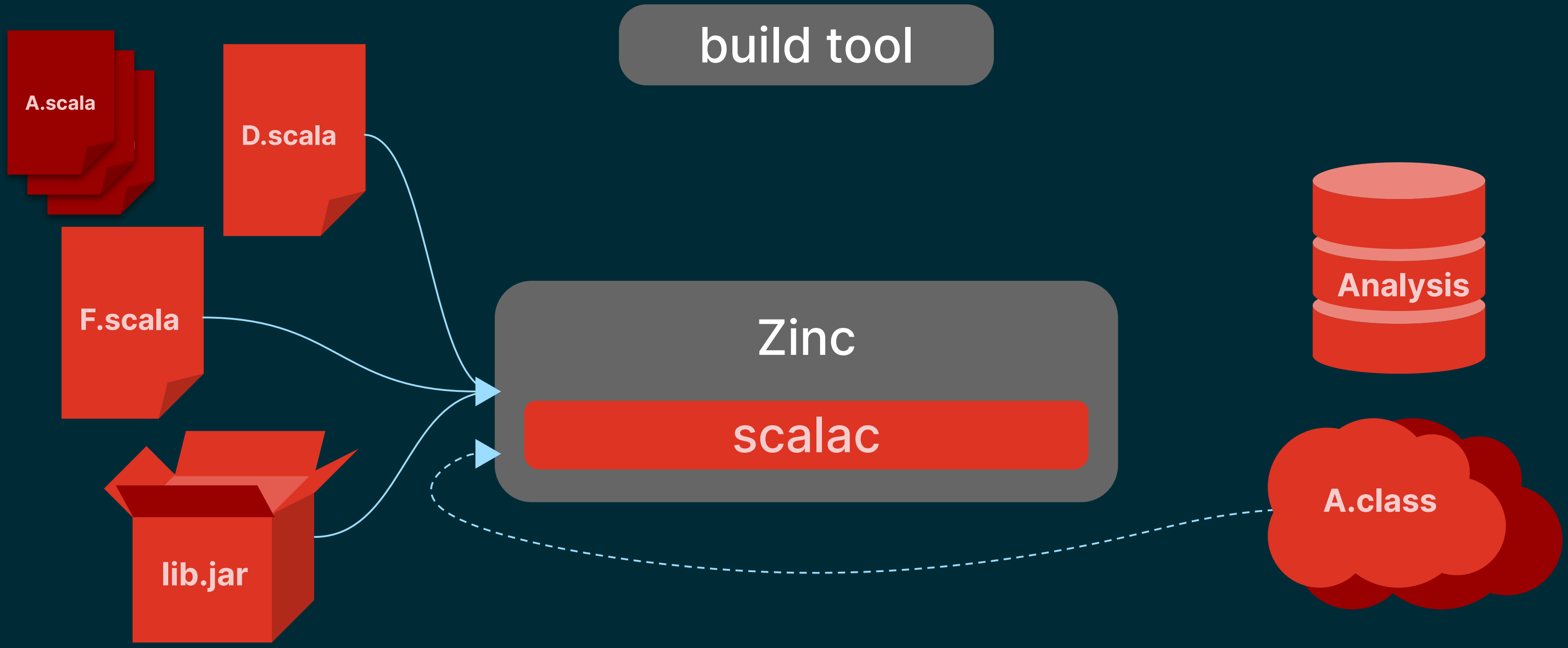
Incremental Compilation



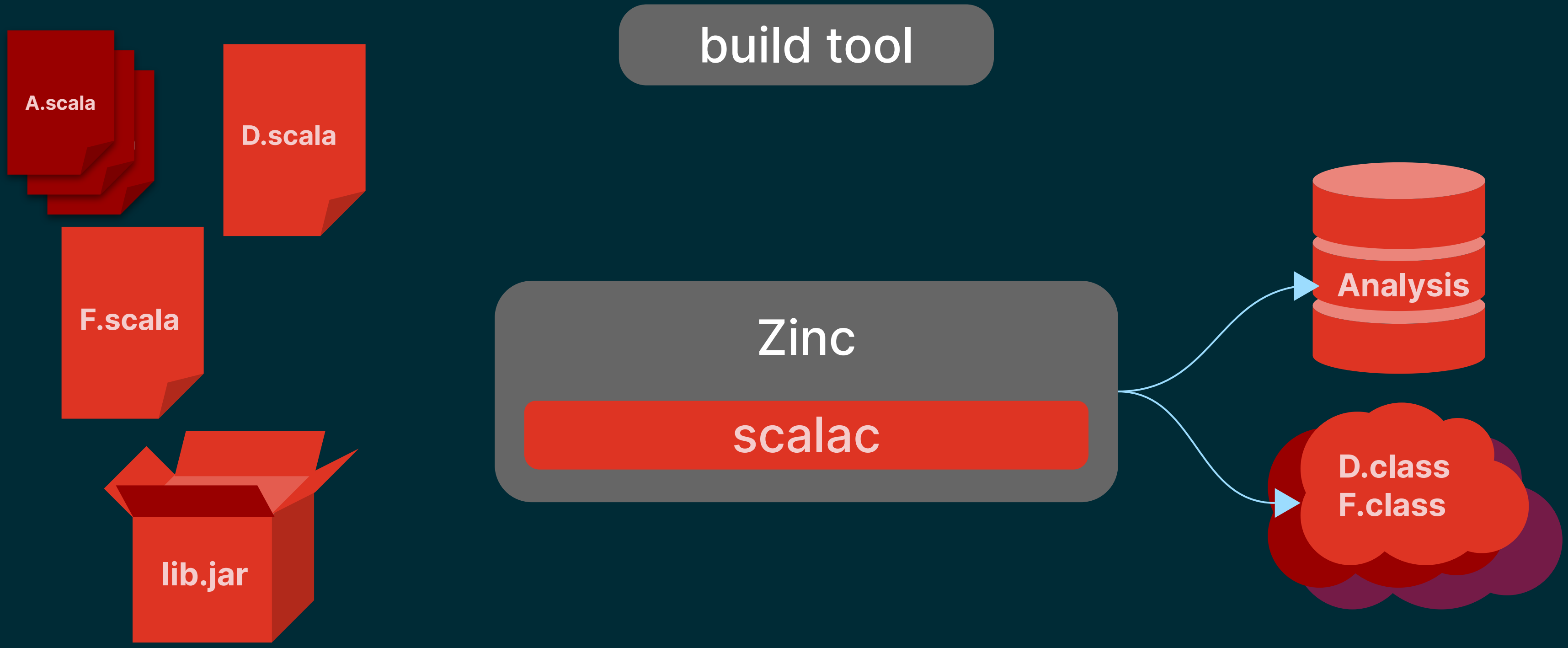
Incremental Compilation


A.scala
D.scala
F.scala
lib.jar
build tool
“recompile D.scala and F.scala, they depend on definitions of A.scala that changed.”
Zinc
scalac
Analysis
A.class

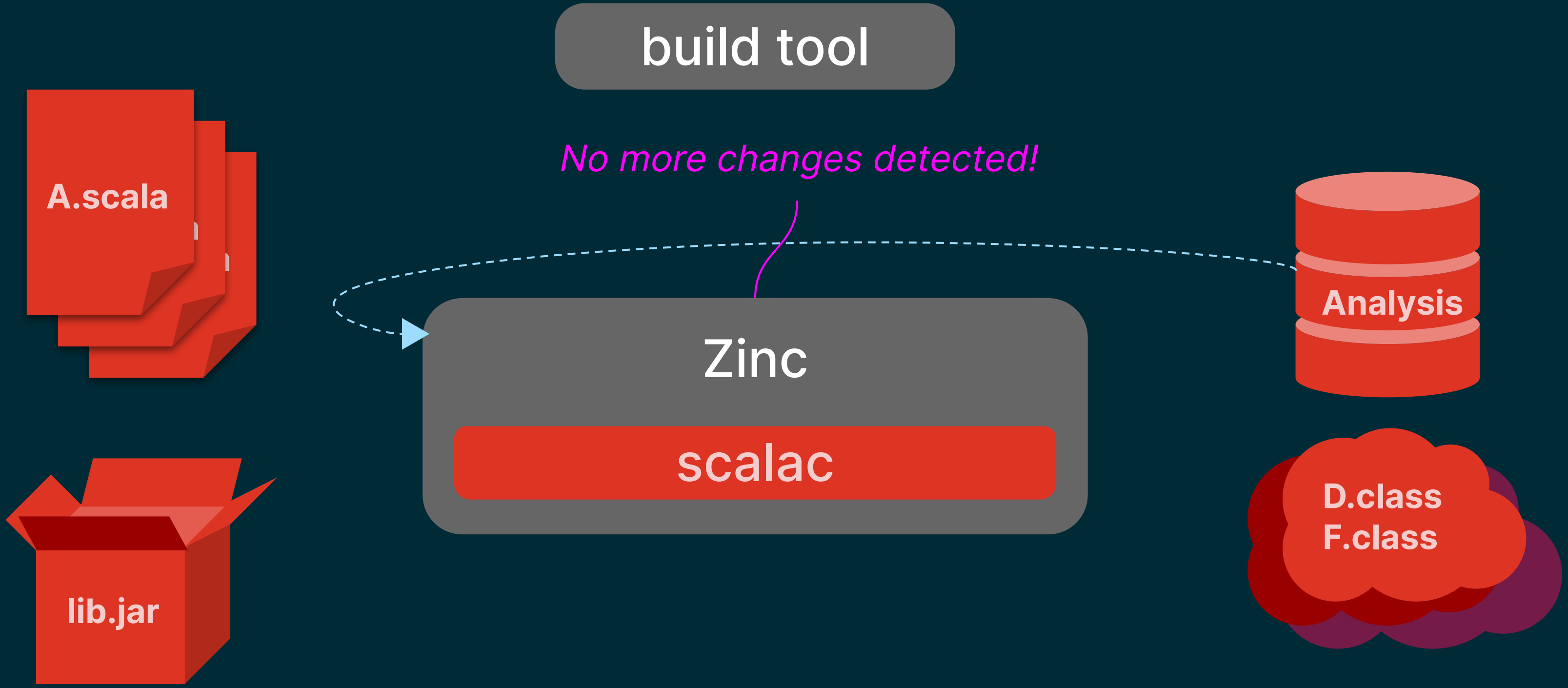
Incremental Compilation



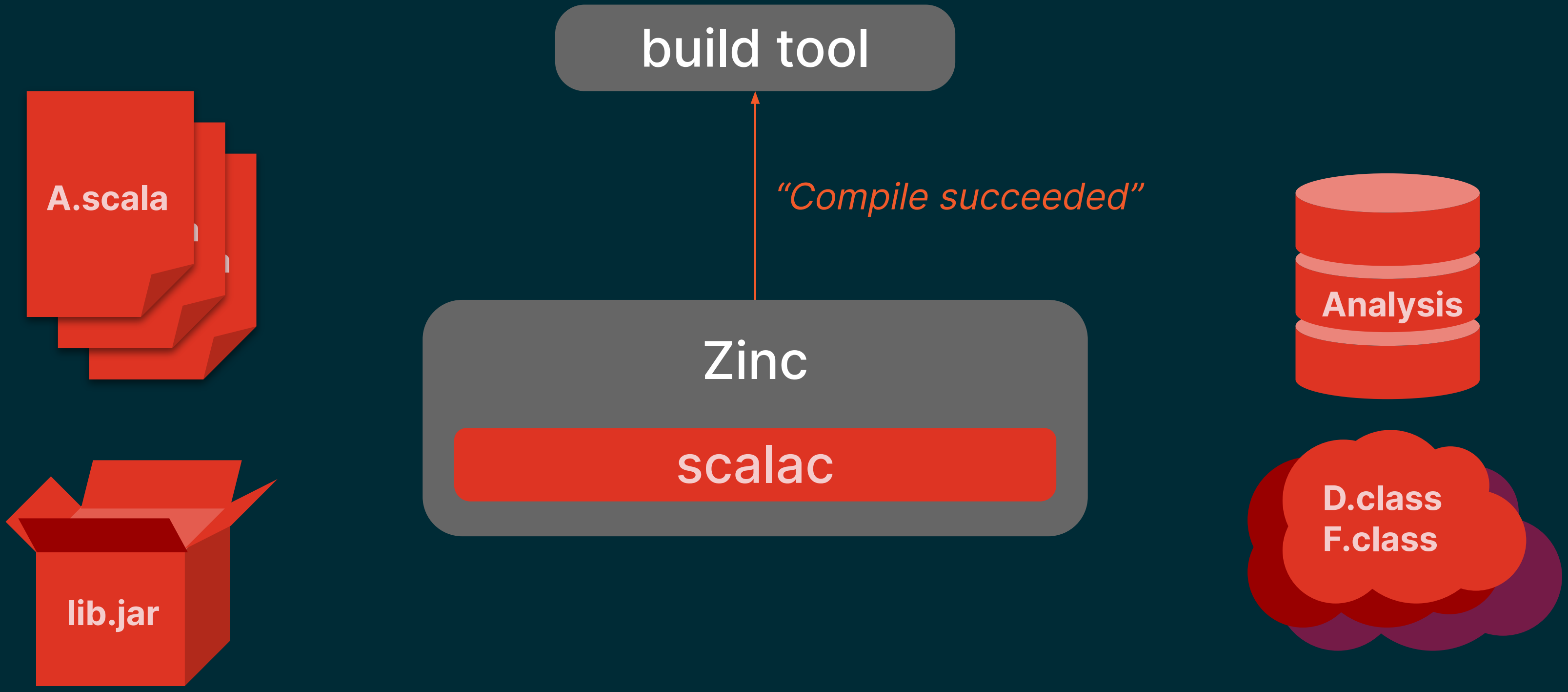
Incremental Compilation



Incremental Compilation



Incremental Compilation



Incremental Compilation

Stamps — *timestamps, file hashes*

APIs — *tree data structure, representing signatures*

Dependencies — *pairs of used-name, file of origin*

Incremental Compilation

Stamps

timestamps, file hashes

*"**A.scala** has different bytes than the last time I saw it, it should be recompiled."*

Incremental Compilation

APIs

tree data structure, representing signatures

```
APIs in A.scala:
class A:
  def foo: Int
  def foo(x: Int): Int
  def foo(x: Int, y: String): Int
  val bar: Boolean
```

Incremental Compilation

computed by Zinc

APIs - Name Hashes

```
class A defines names:
  foo = 0x8523a23
  bar = 0x4d65e65
```

aggregate all "foo" API

aggregate all "bar" API

Incremental Compilation

APIs - Name Hashes

```
class A defines names:
  foo = 0xfb191c7
  bar = 0x4d65e65
```

*"Some definition **A.foo** has a **changed API**"*

Incremental Compilation

Dependencies

pairs of used-name, class of origin

```
class D inherits from class A
class F uses name foo
class F uses a member of class A
```

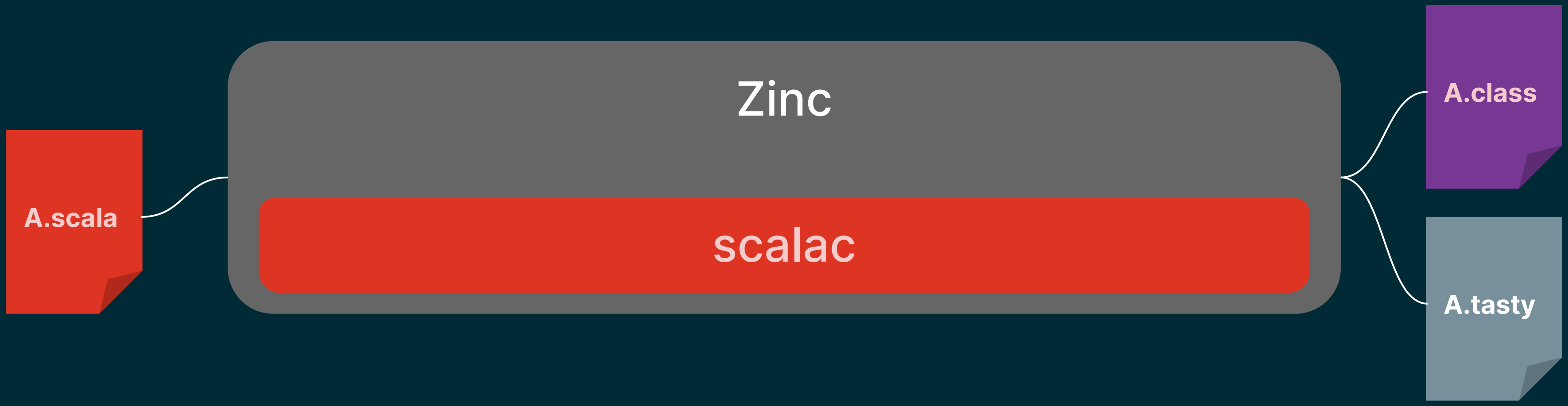
*"both **class D** and **class F** depend on changed API's of **class A**!"*

Incremental Compilation

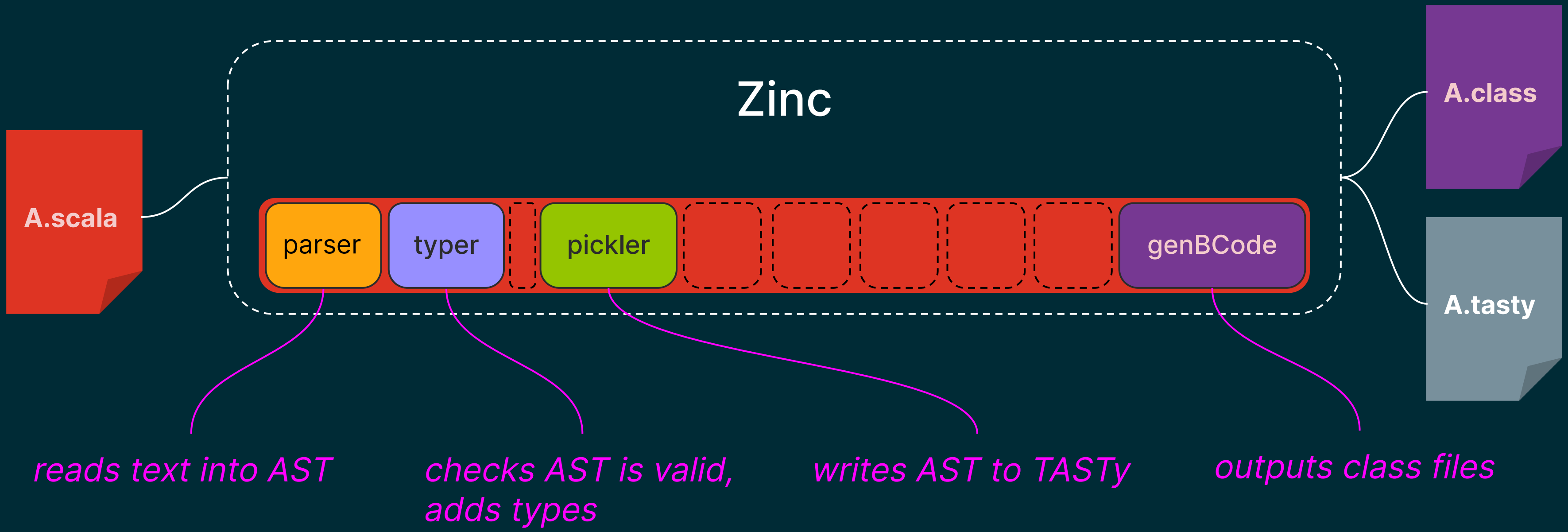
Summary

The combination of **stamps**, **name hashes** and **dependencies** are sufficient to maximise **performance** and **correctness** of the name hashing algorithm.

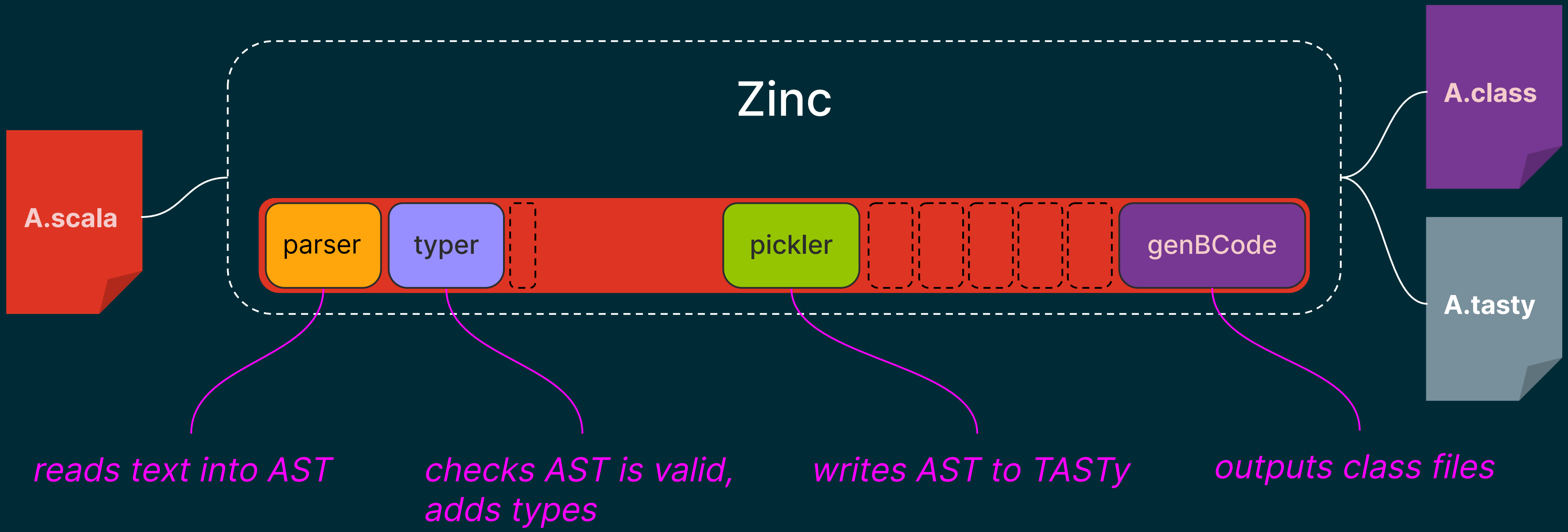
Incremental Compilation



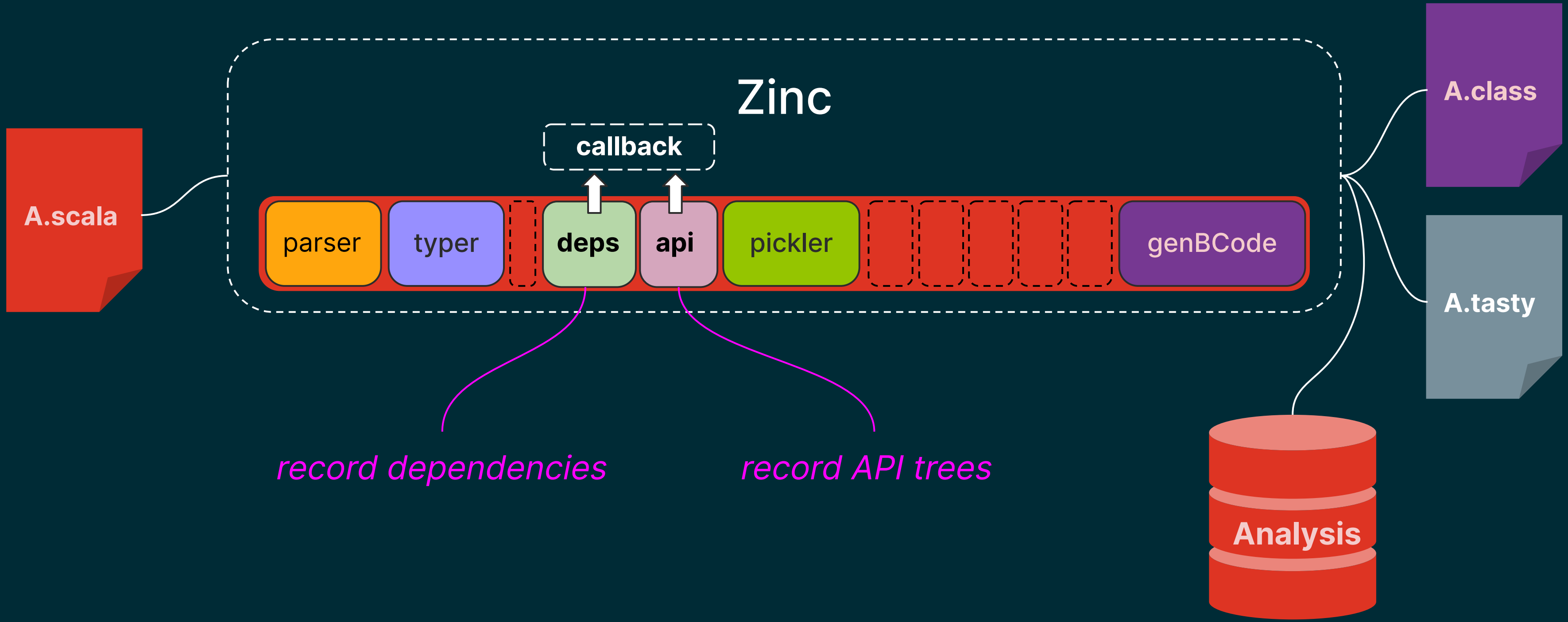
Incremental Compilation



Incremental Compilation



Incremental Compilation

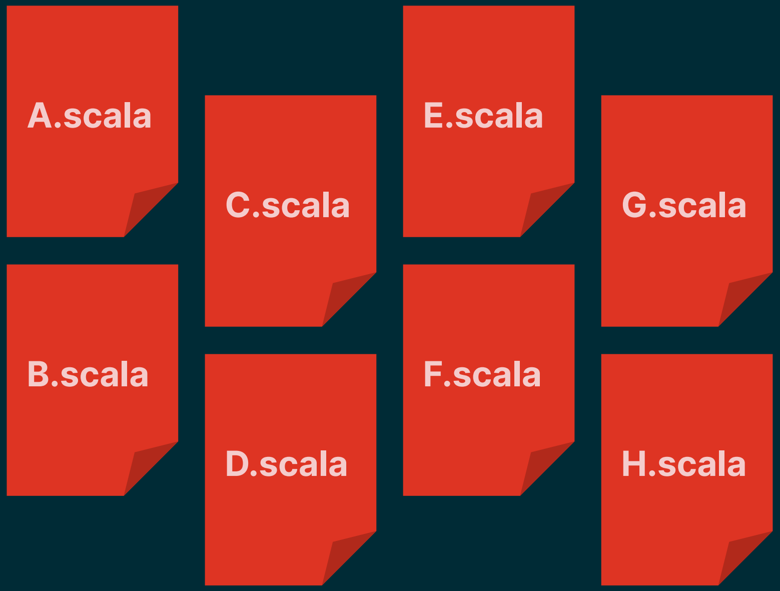


Multi-project builds

structure your project as a collection of **modular libraries** that cooperate to form a **cohesive whole**.

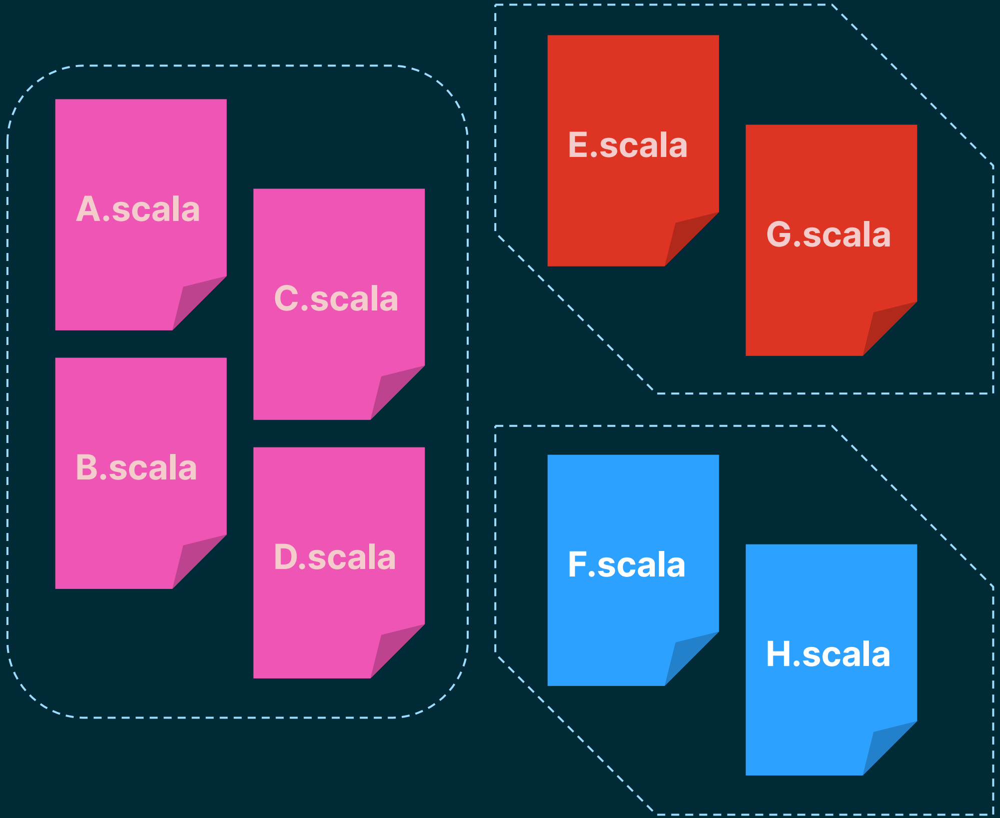
Multi-project Builds

Split up the source files into modular groups



Multi-project Builds

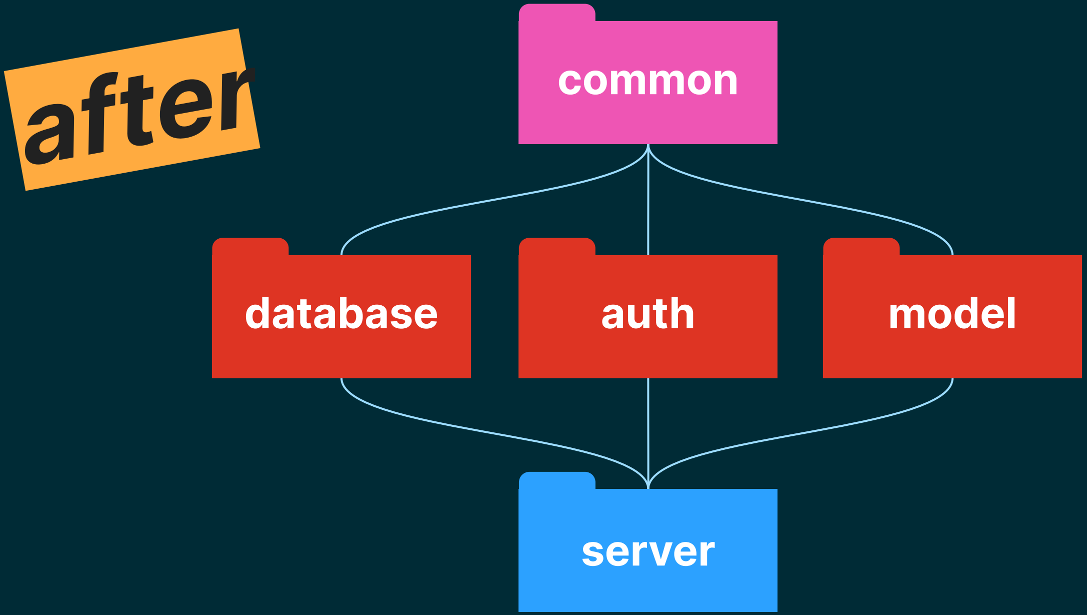
each group can be compiled in separate stages



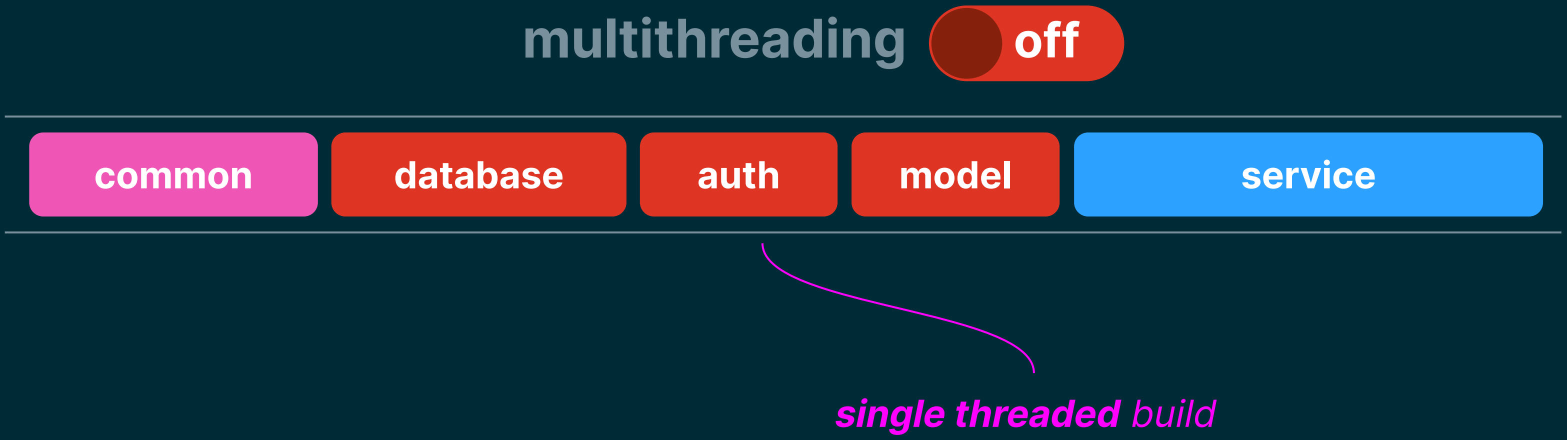
Multi-project Builds

before
webservice

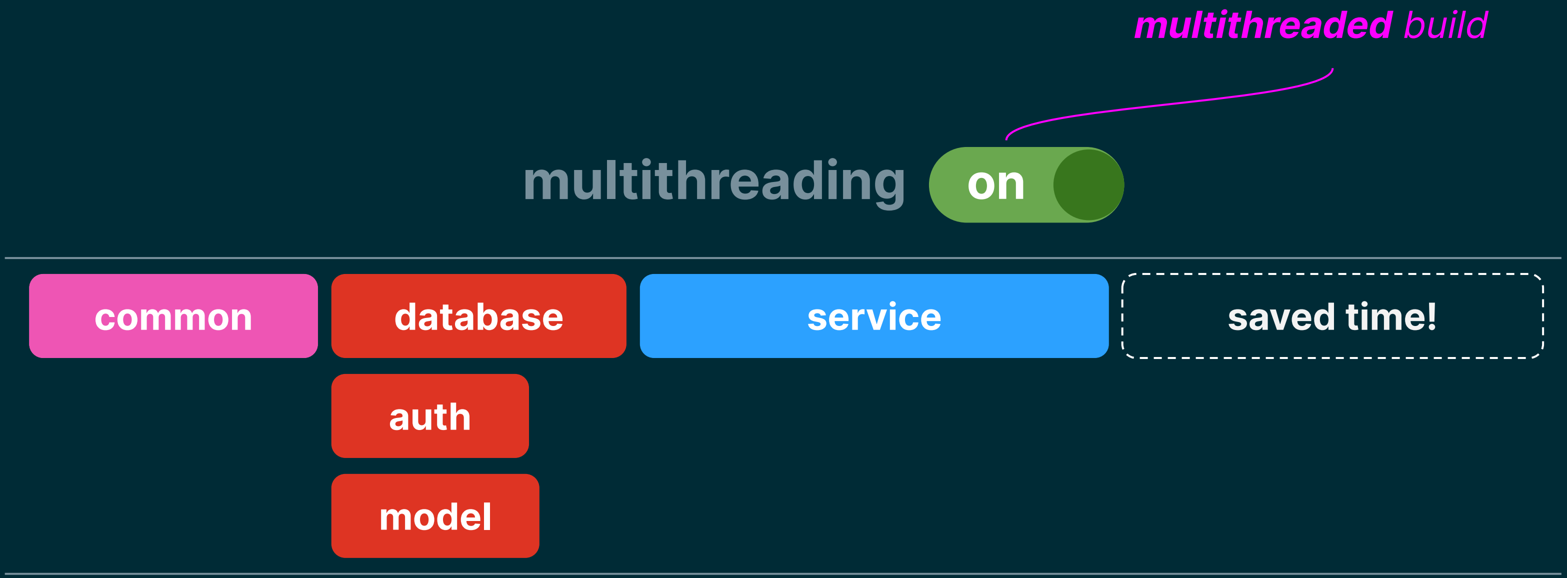
Multi-project Builds



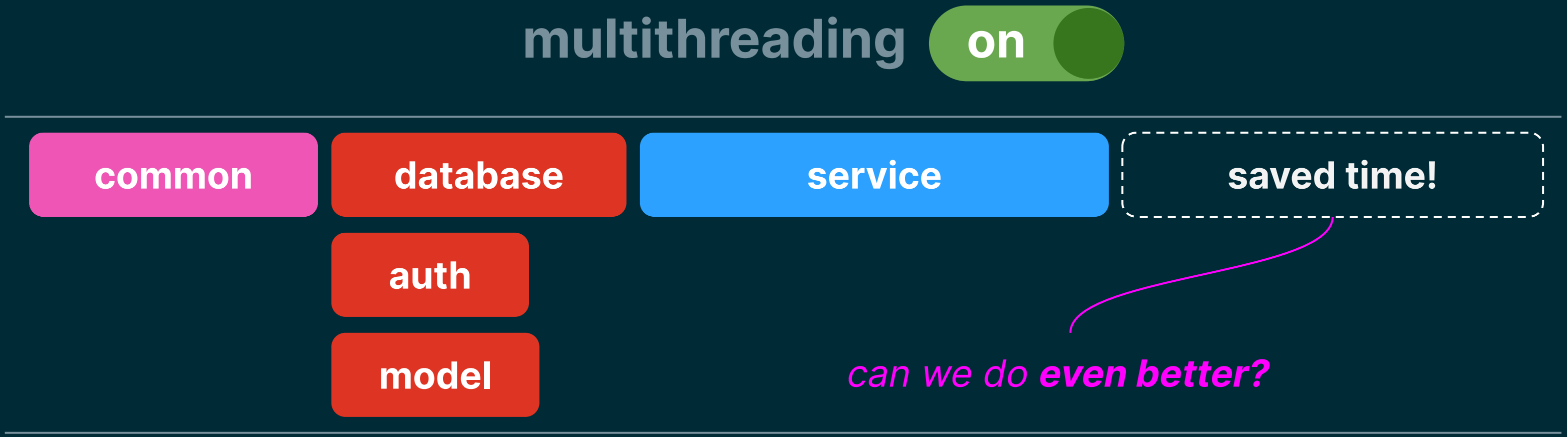
Multi-project Builds



Multi-project Builds



Multi-project Builds



Pipelined Builds

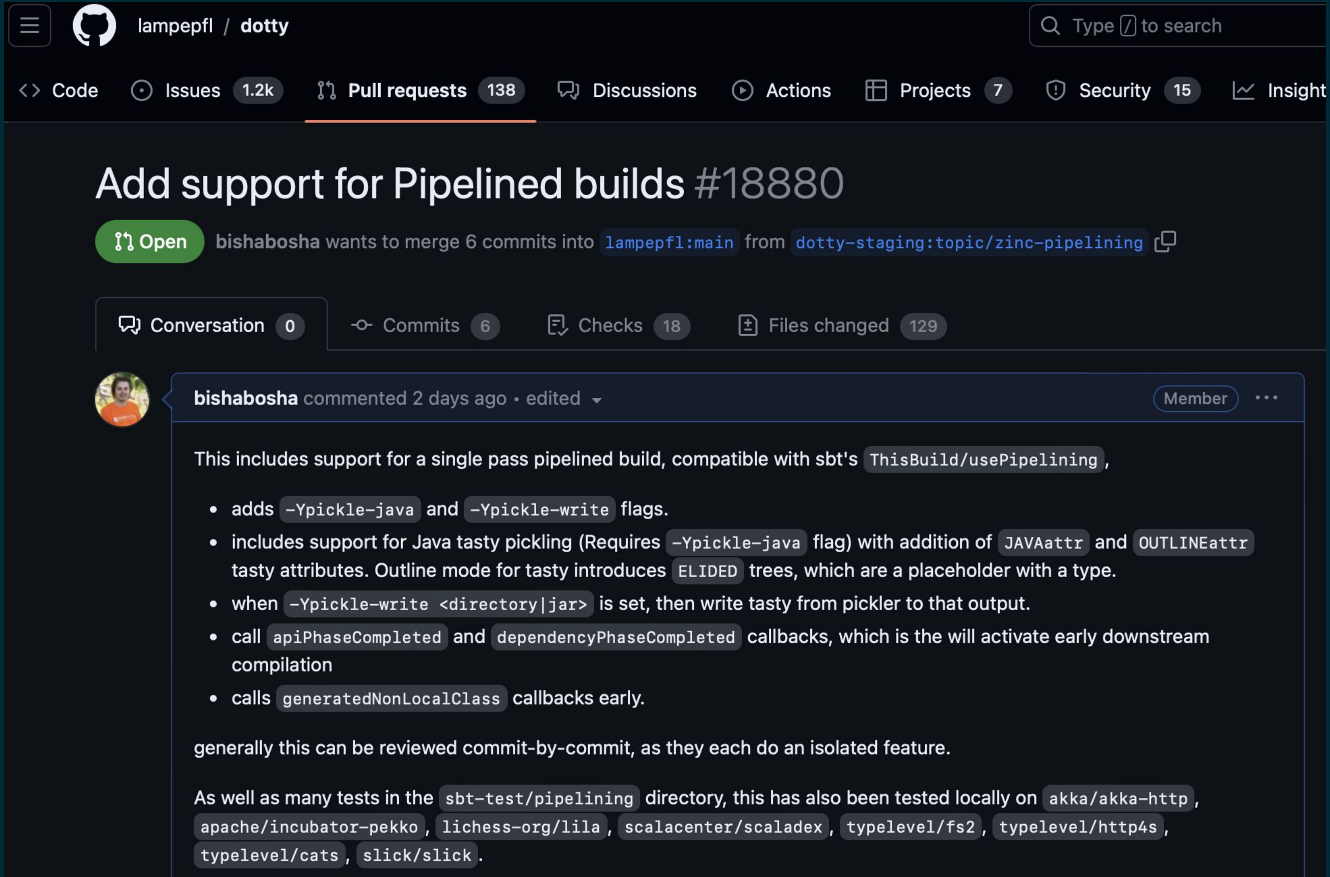
When possible, **begin downstream compilation** before the **upstream completes**.

Pipelined Builds

Prior Work

- Morgan Stanley OBT (optimus platform)
- Bloop with Zinc fork (Jorge Vicente Cantero)
- Experimental support **today** in **sbt**

Pipelined Builds



lampepfl / dotty

Code | Issues 1.2k | Pull requests 138 | Discussions | Actions | Projects 7 | Security 15 | Insight

Add support for Pipelined builds #18880

Open bishabosha wants to merge 6 commits into `lampepfl:main` from `dotty-staging:topic/zinc-pipelining`

Conversation 0 | Commits 6 | Checks 18 | Files changed 129

bishabosha commented 2 days ago • edited — Member

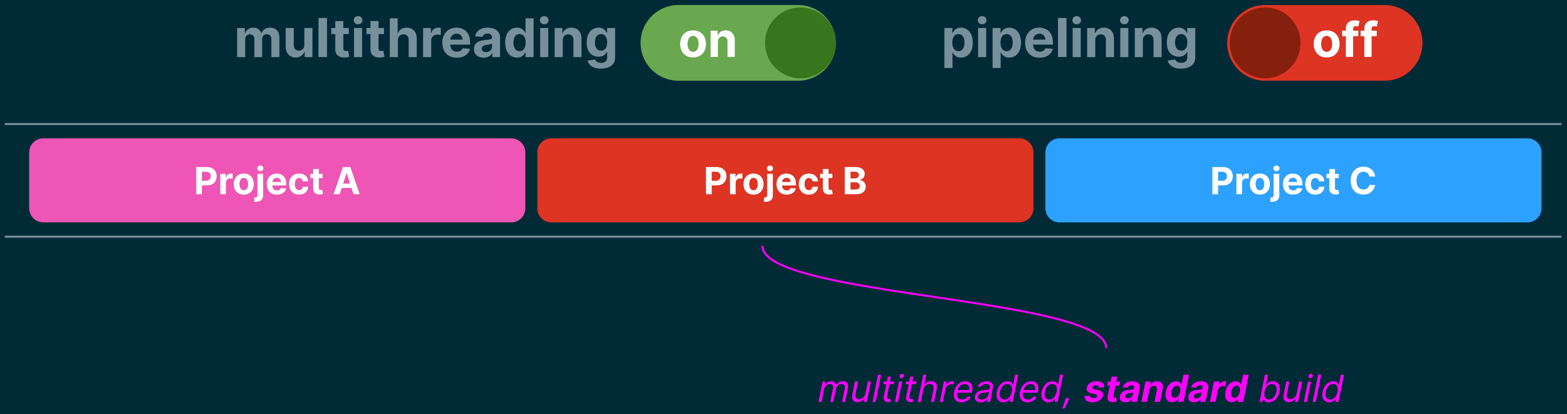
This includes support for a single pass pipelined build, compatible with sbt's `ThisBuild/usePipelining`,

- adds `-Ypickle-java` and `-Ypickle-write` flags.
- includes support for Java tasty pickling (Requires `-Ypickle-java` flag) with addition of `JAVAattr` and `OUTLINEattr` tasty attributes. Outline mode for tasty introduces `ELIDED` trees, which are a placeholder with a type.
- when `-Ypickle-write <directory|jar>` is set, then write tasty from pickler to that output.
- call `apiPhaseCompleted` and `dependencyPhaseCompleted` callbacks, which is the will activate early downstream compilation
- calls `generatedNonLocalClass` callbacks early.

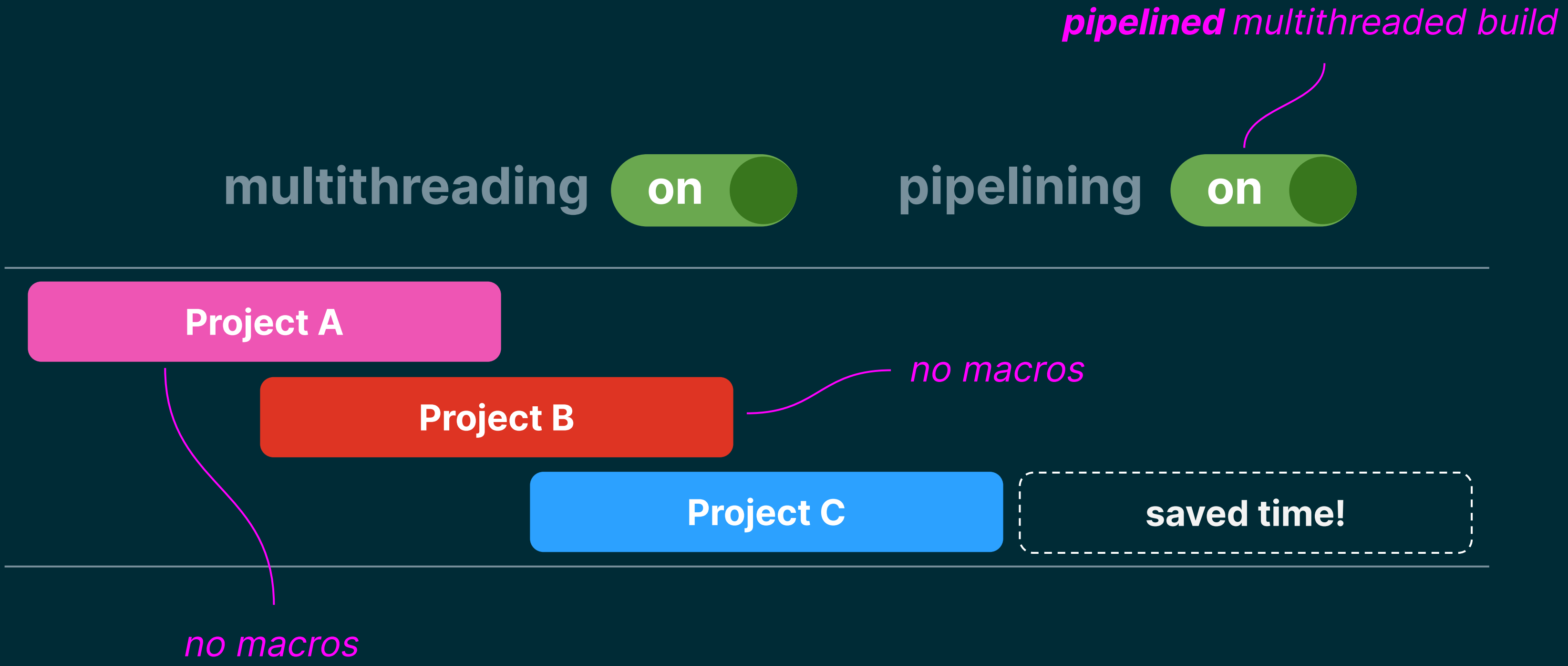
generally this can be reviewed commit-by-commit, as they each do an isolated feature.

As well as many tests in the `sbt-test/pipelining` directory, this has also been tested locally on `akka/akka-http`, `apache/incubator-pekko`, `lichess-org/lila`, `scalacenter/scaladex`, `typelevel/fs2`, `typelevel/http4s`, `typelevel/cats`, `slick/slick`.

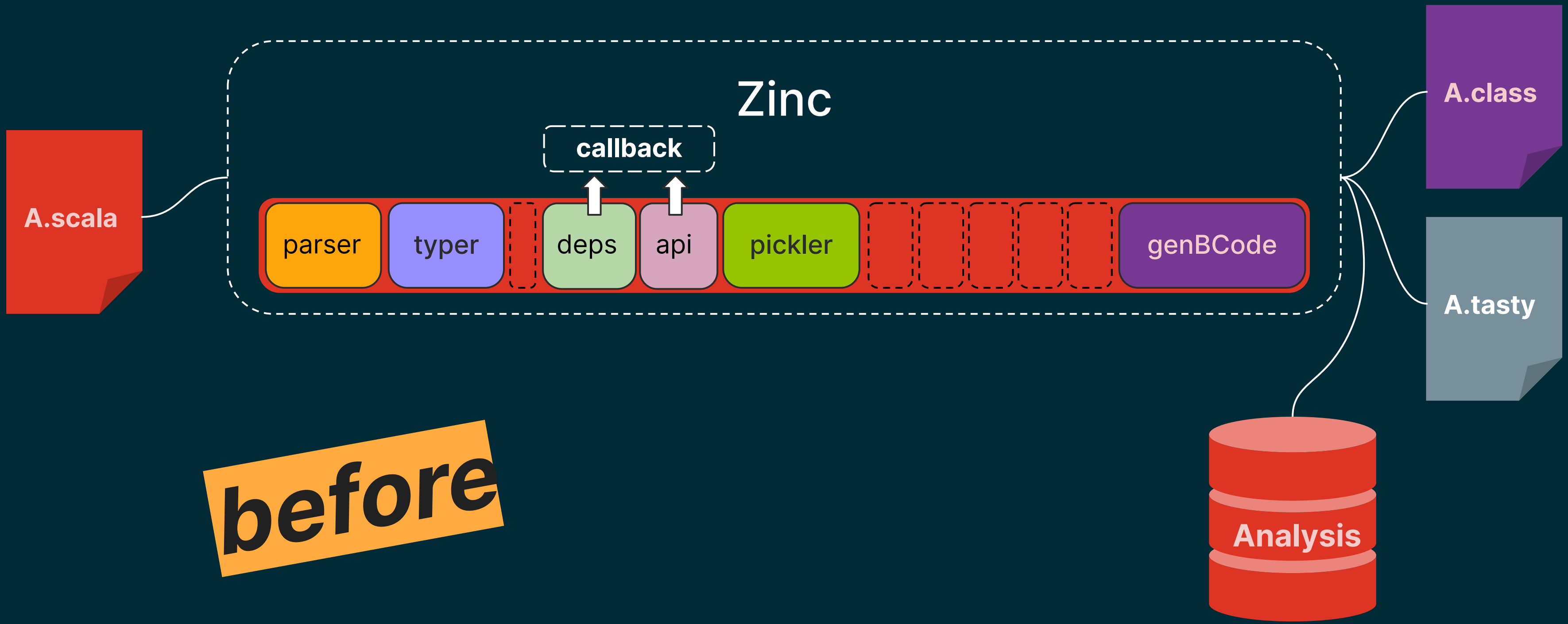
Pipelined Builds



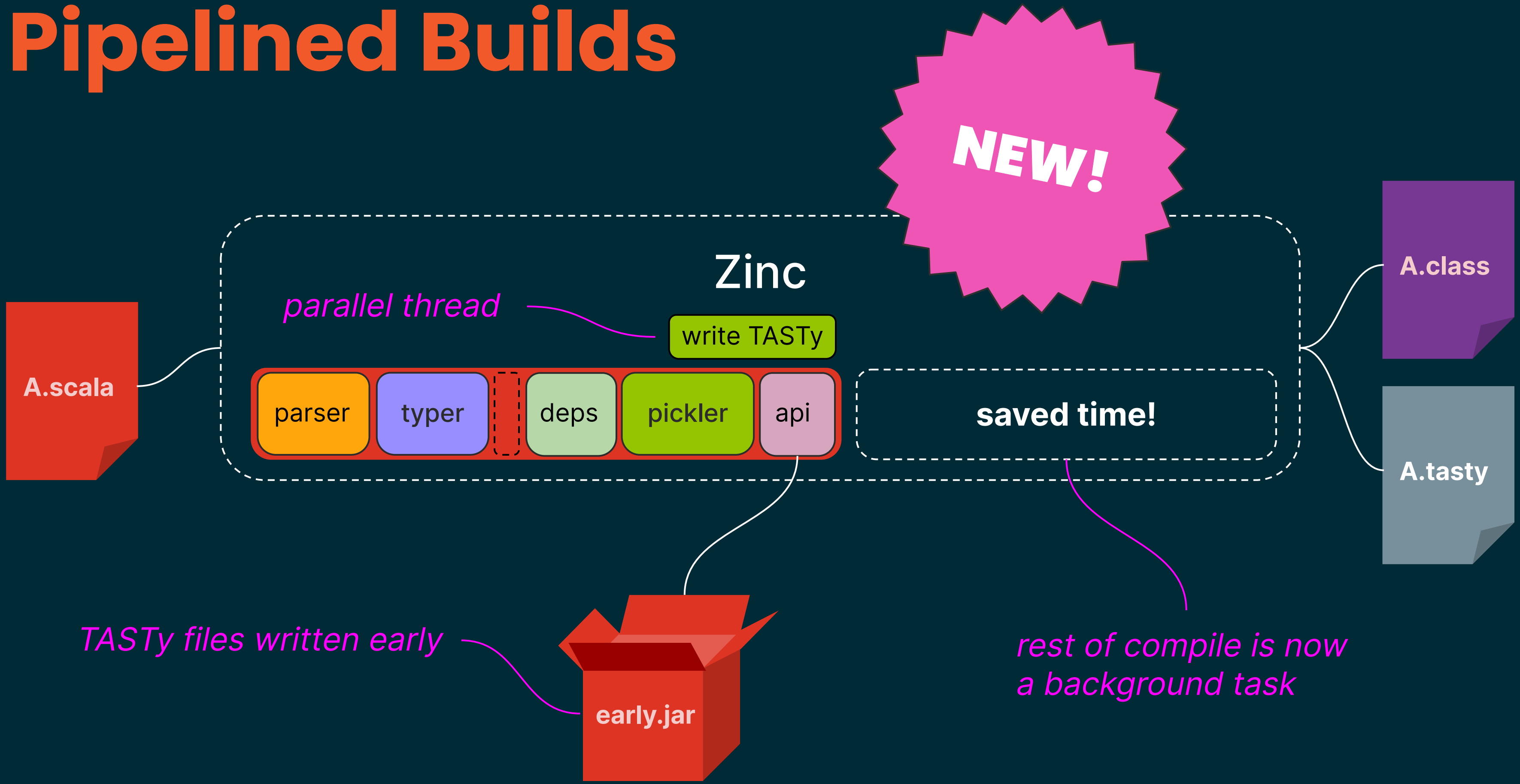
Pipelined Builds



Pipelined Builds



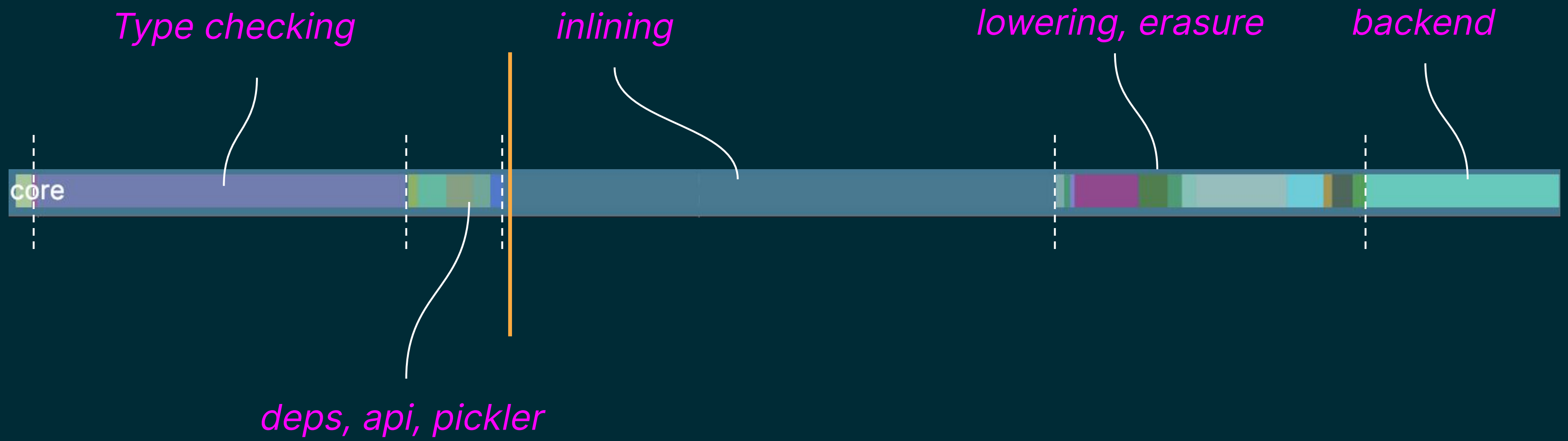
Pipelined Builds



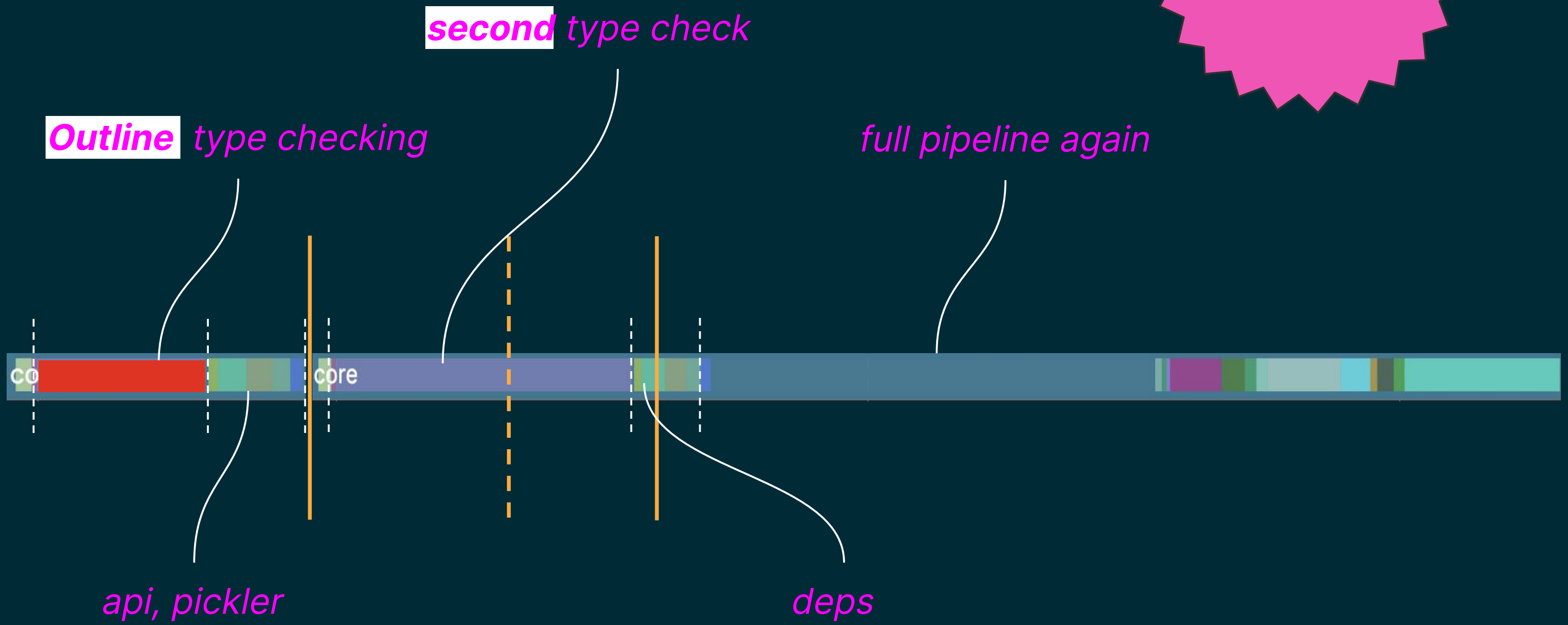
outline compile

Compile in two passes, first **quick to signatures**, and a **parallel second pass**.

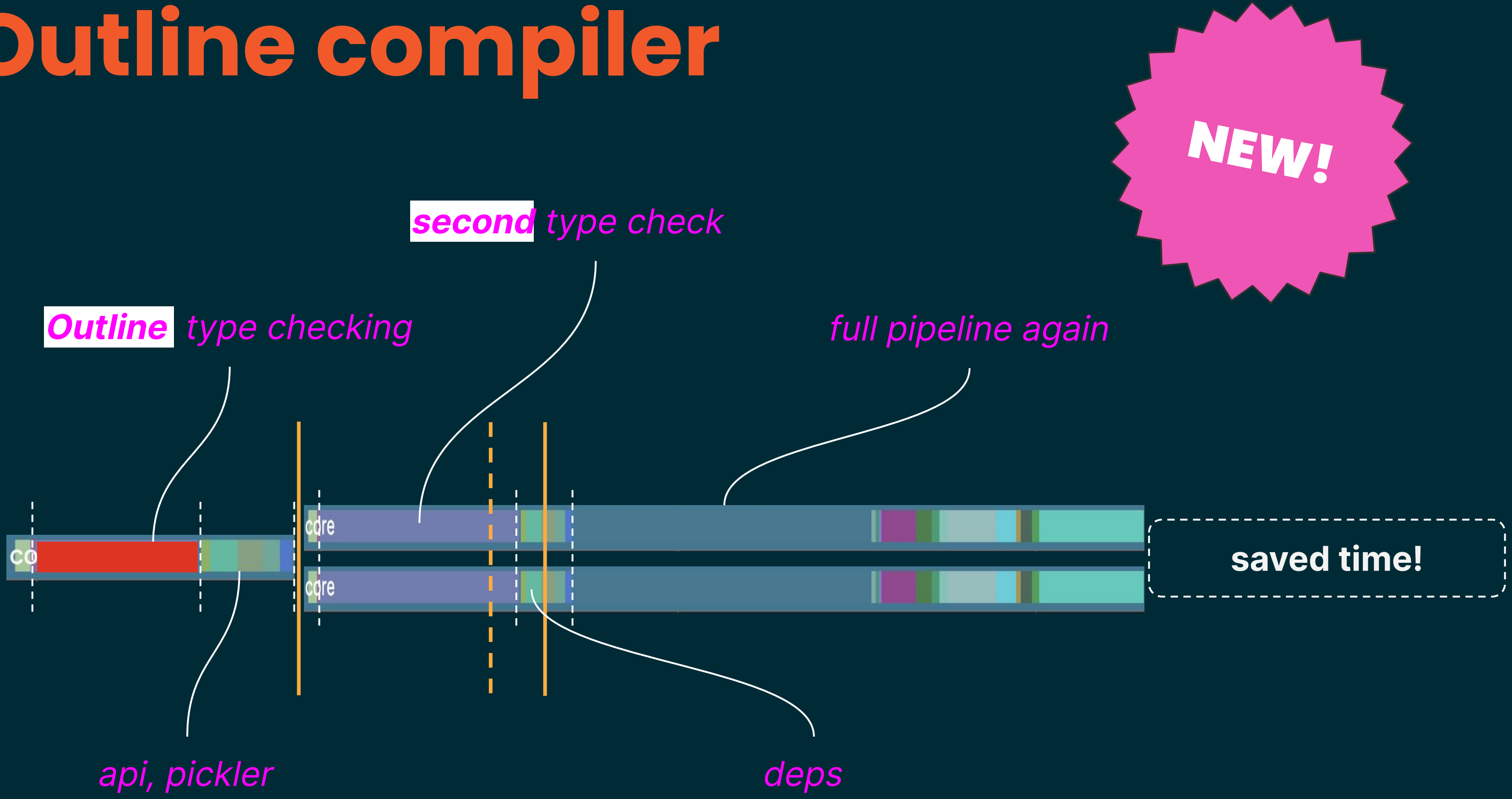
standard compiler



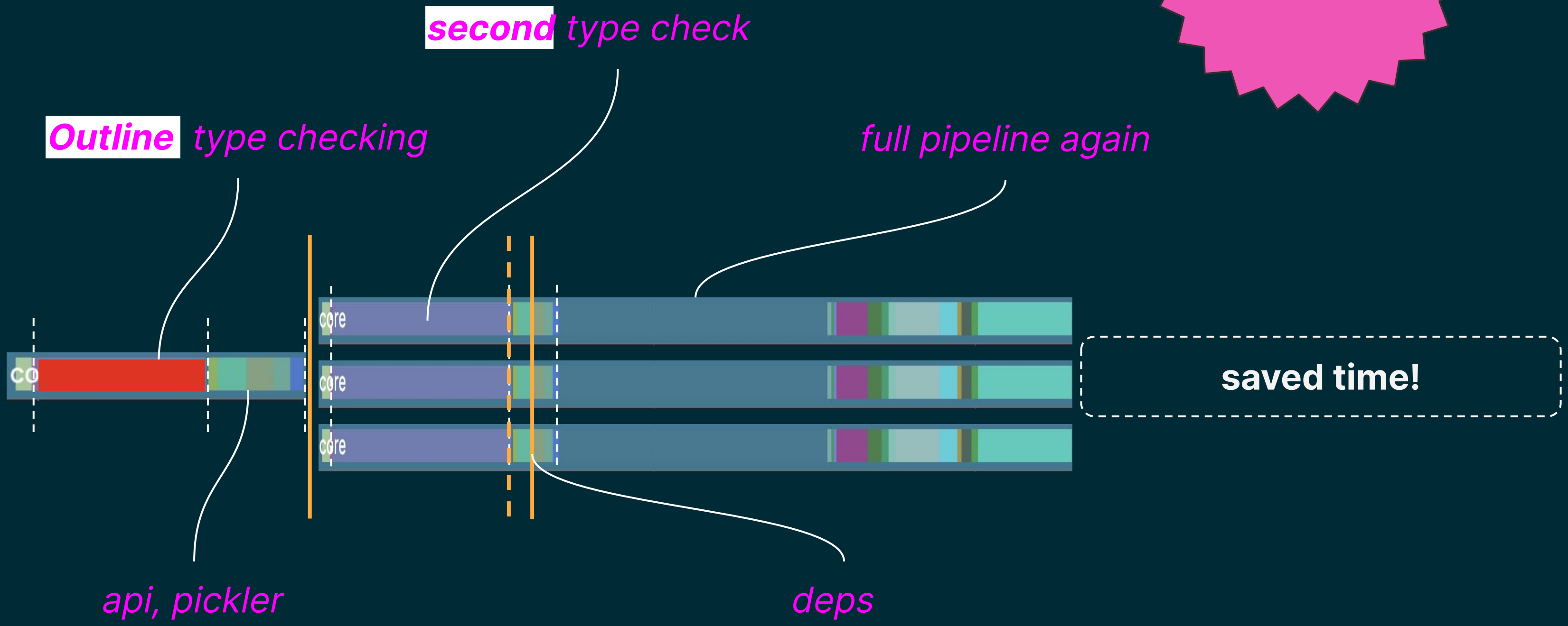
Outline compiler



Outline compiler



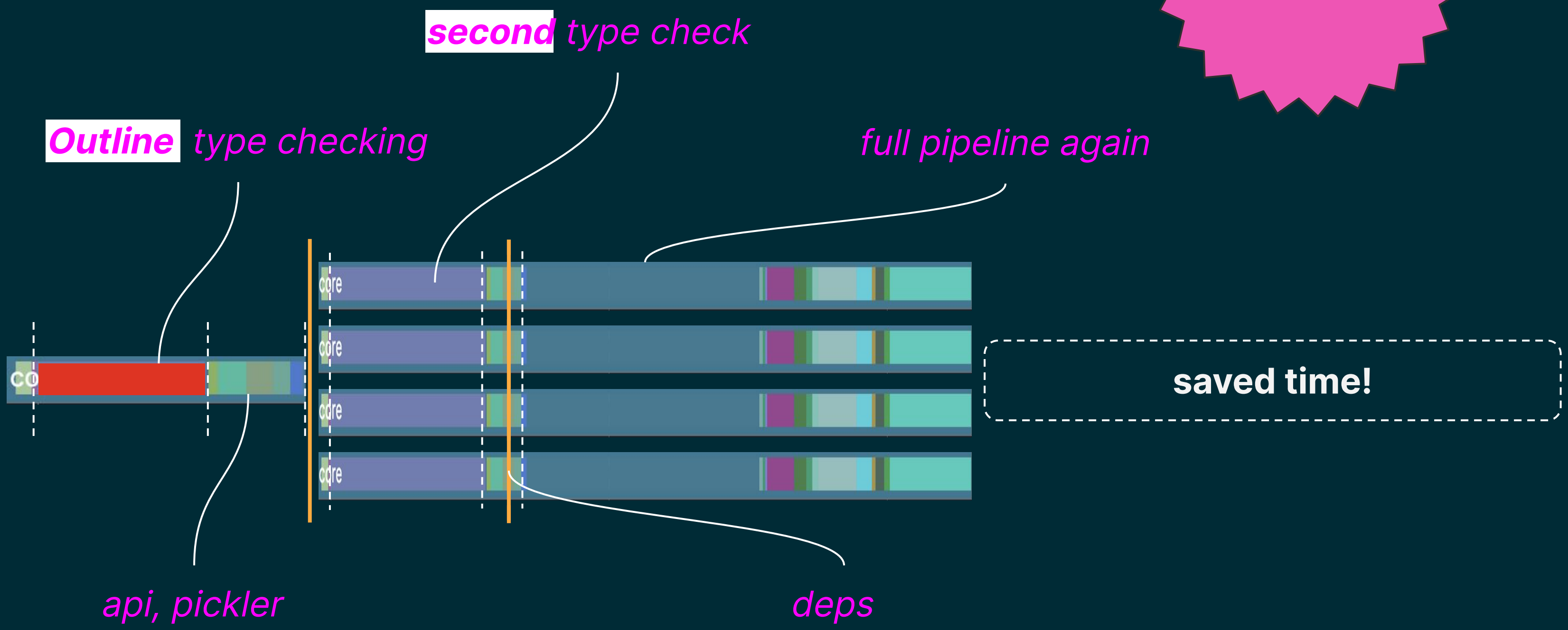
Outline compiler



NEW!

Outline compiler

NEW!



Takeaways

What can you do to **improve build times**?

Takeaways

Tip No. 1

Use small files!

A more granular dependency graph avoids unnecessary recompilation

Takeaways

Tip No. 2

Split apps into smaller projects!

With a more granular project graph, you can introduce parallelism.

Takeaways

Tip No. 3

Don’t make projects too small!

Thread starvation, duplicate work

Takeaways

Tip No. 4

Try out pipelining!

$\underbrace{\hphantom{pipelining!}}$

The faster the better, right?!

Demo

Benchmarks

Enough! **show me the numbers**!

Benchmarks

Testing on MacBook Pro 2019

(i9 8-core 2.3GHz 16GB RAM)

From cold sbt start:

- `clean; compile` 2x to warm up
- then take mean time of next 7 cycles.

Benchmarks – pipelining

lichess-org/lila

308,829 LOC

The key takeaway seems to be that you trade **time** overall for **peak memory**.

72s ◆**6GB**

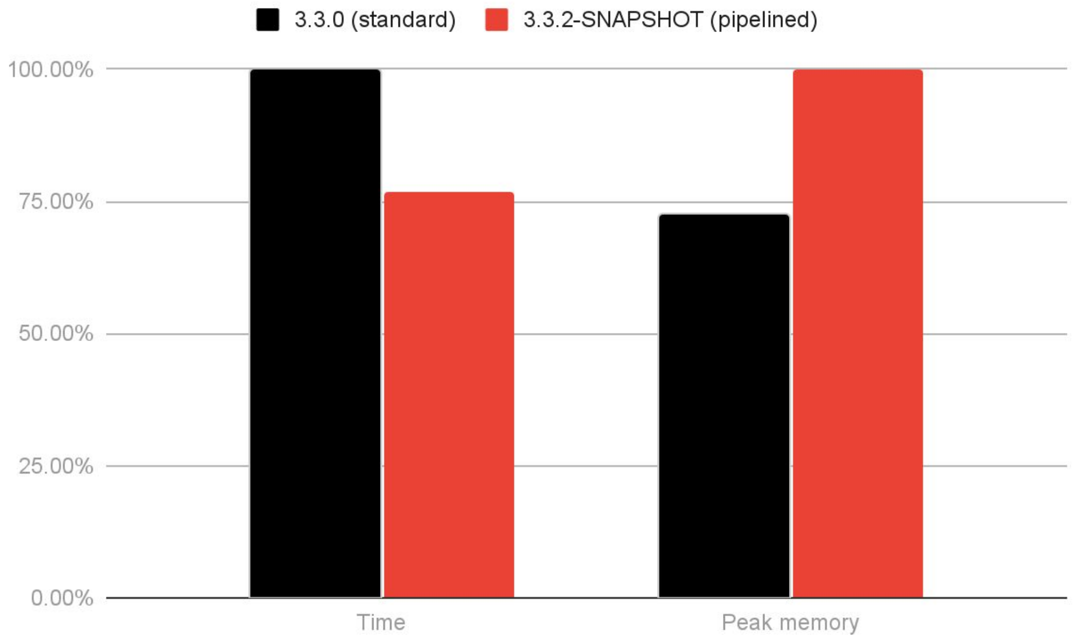
■ 3.3.0 (standard)

5600 lines/s

◆**55s** **8.3GB**

■ 3.3.2-SNAPSHOT (pipelined)

“clean compile” time & memory



Benchmarks – pipelining

Other projects

Your **mileage may vary**

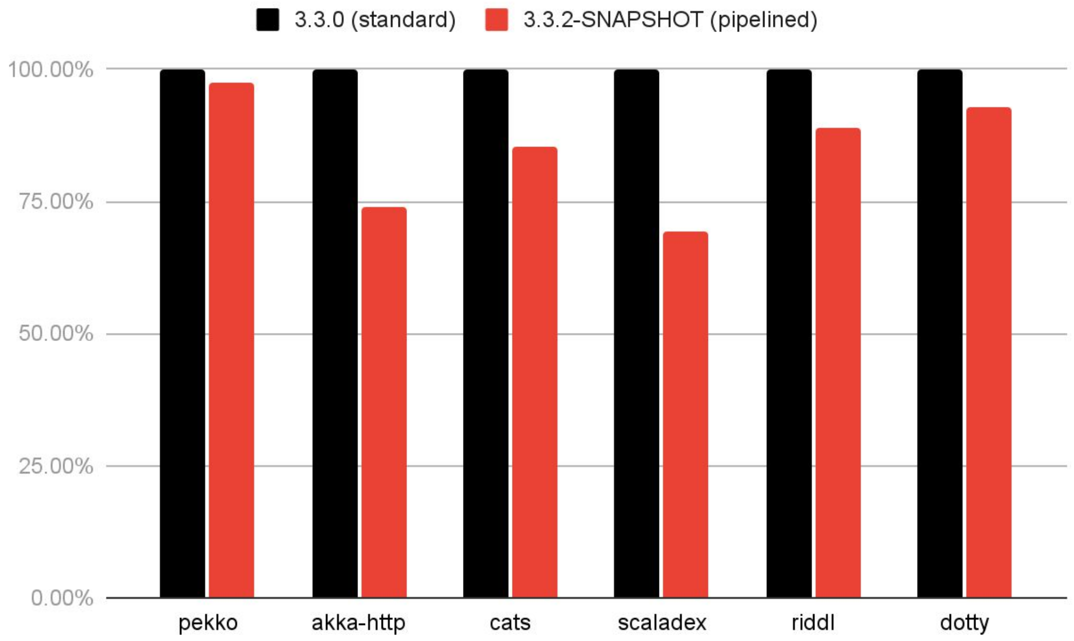
Scaladex

31% improved

🟥 3.3.2-SNAPSHOT (pipelined)

Time to finish

"clean compile" time



Benchmarks – outline compile

lampepfl/dotty

142,000 LOC

33s 4900 lines/s

■ 3.3.0 (single pass)

20s 7500 lines/s

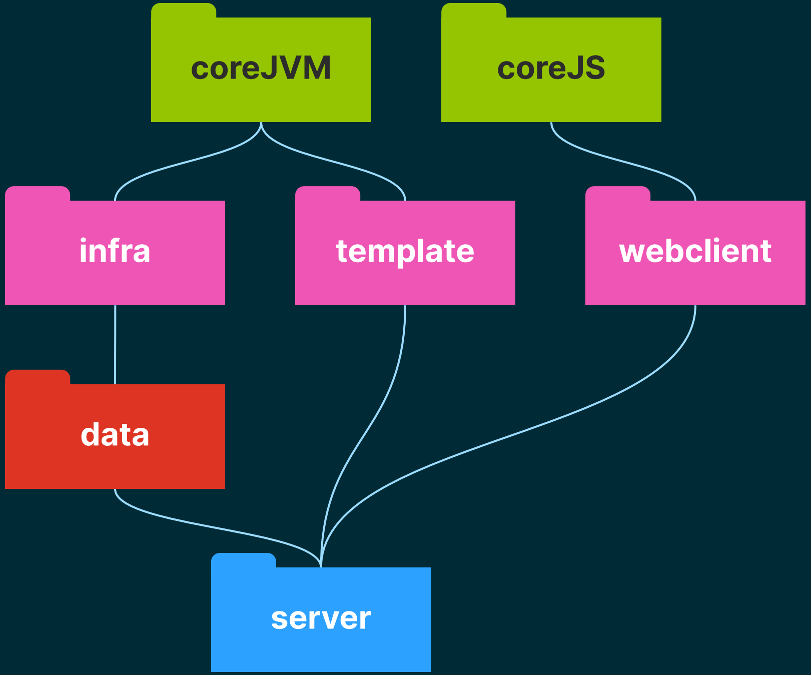
39% improved! we could still do better...

🟥 3.3.2-SNAPSHOT (2-pass)

Benchmarks

Scaladex

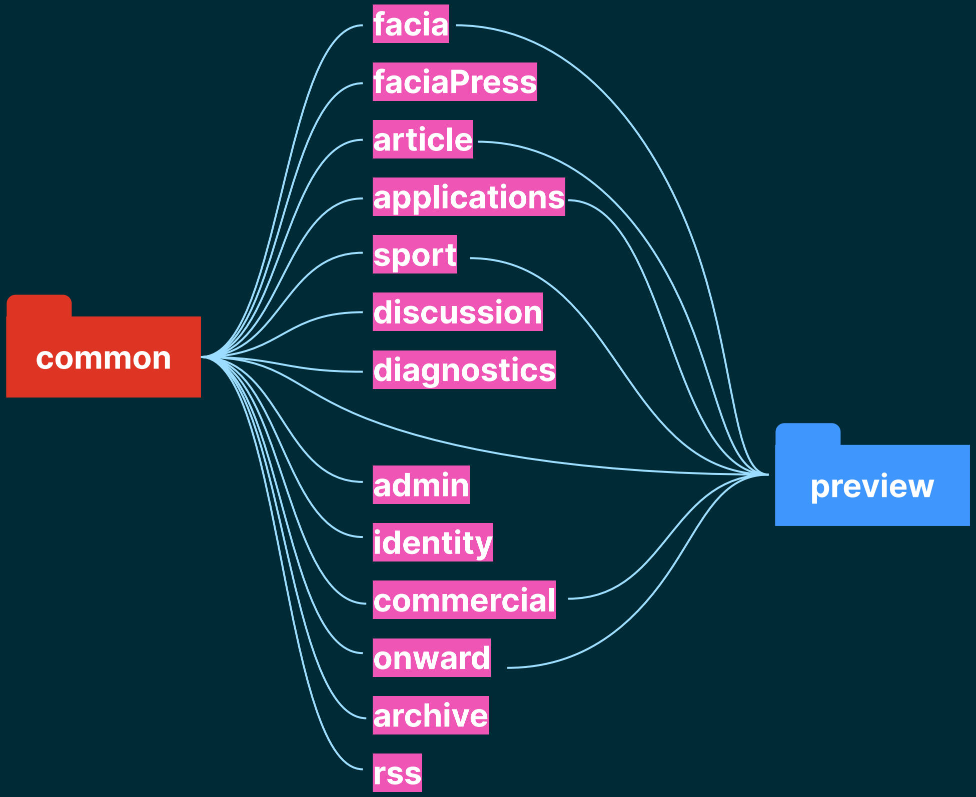
project layout



Benchmarks

guardian frontend

project layout

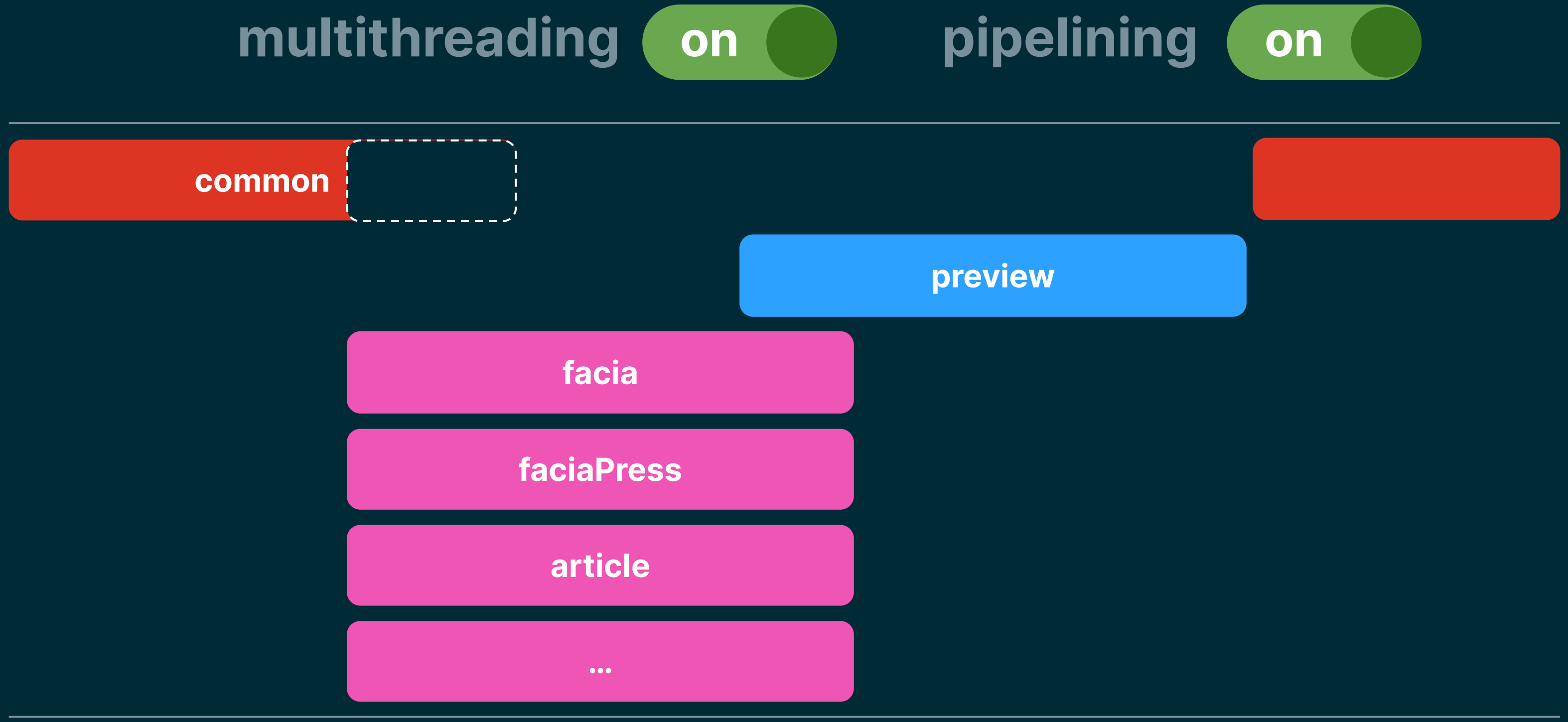


Benchmarks

multithreading on   pipelining off

common	facia	preview
	faciaPress	
	article	
	applications	
	...	

Benchmarks


multithreading
on
pipelining
on
common
preview
facia
faciaPress
article
...