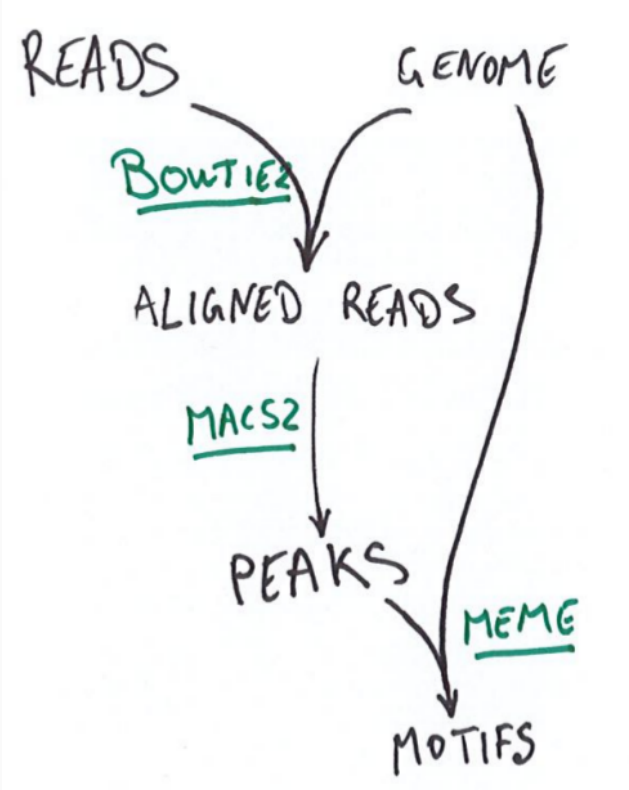
`bistro`: a library to build large-scale workflows in computational biology

Philippe Veber

May 23th, 2019

Laboratoire de Biométrie et Biologie Évolutive

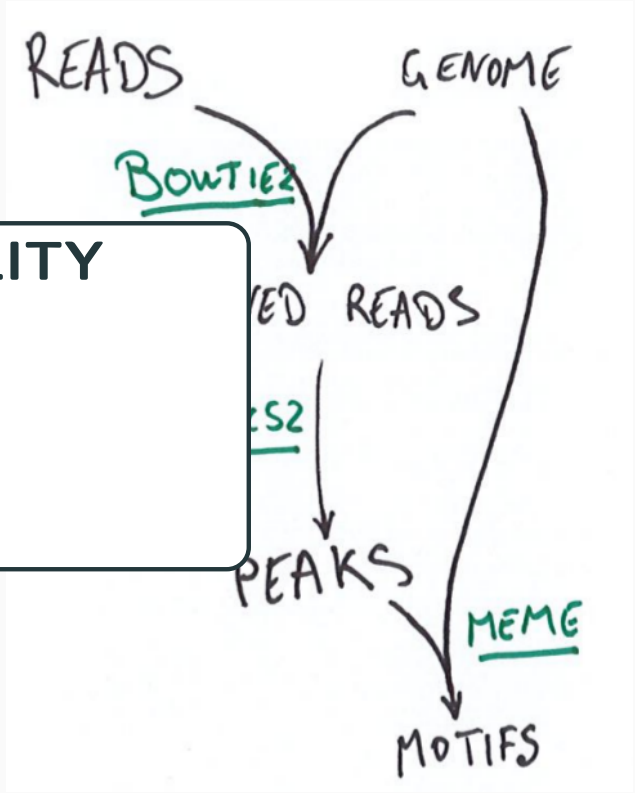
Motivations



Motivations

REPRODUCIBILITY

- version control
- containers
- deployment



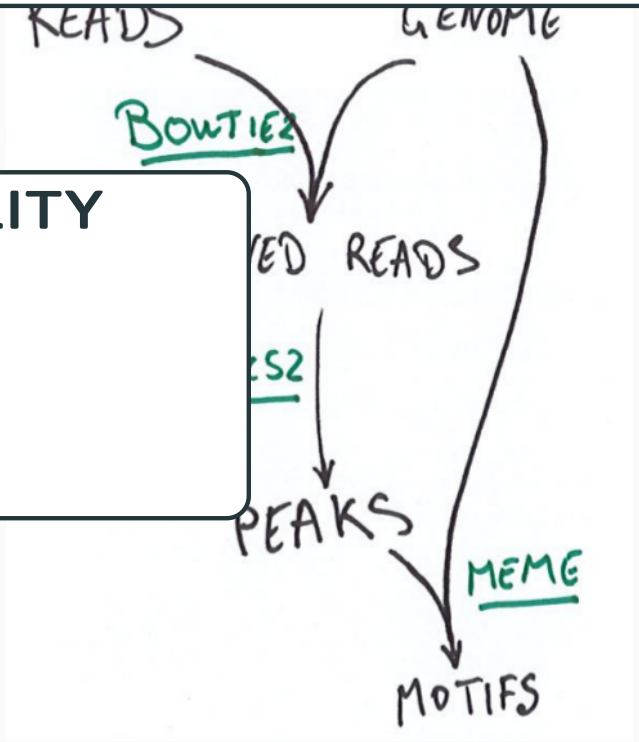
Motivations

EXECUTION

- task distribution
- cluster scheduler

REPRODUCIBILITY

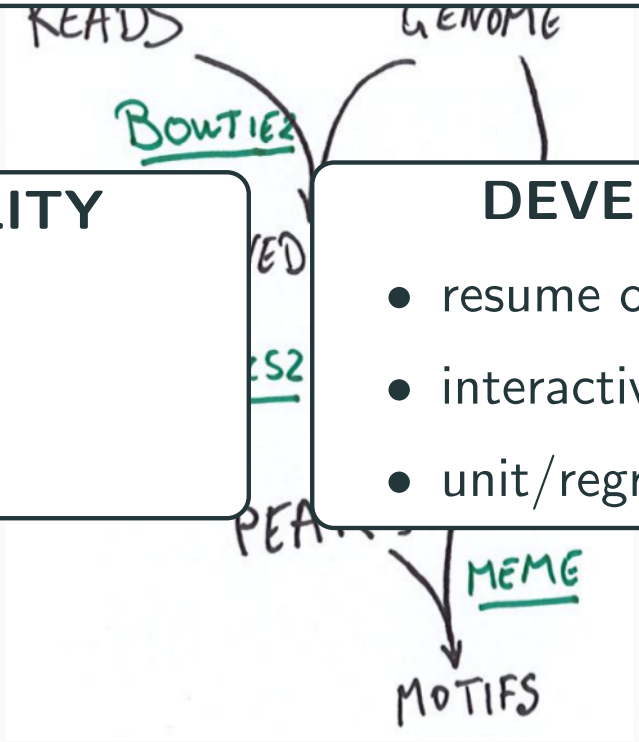
- version control
- containers
- deployment



Motivations

EXECUTION

- task distribution
- cluster scheduler



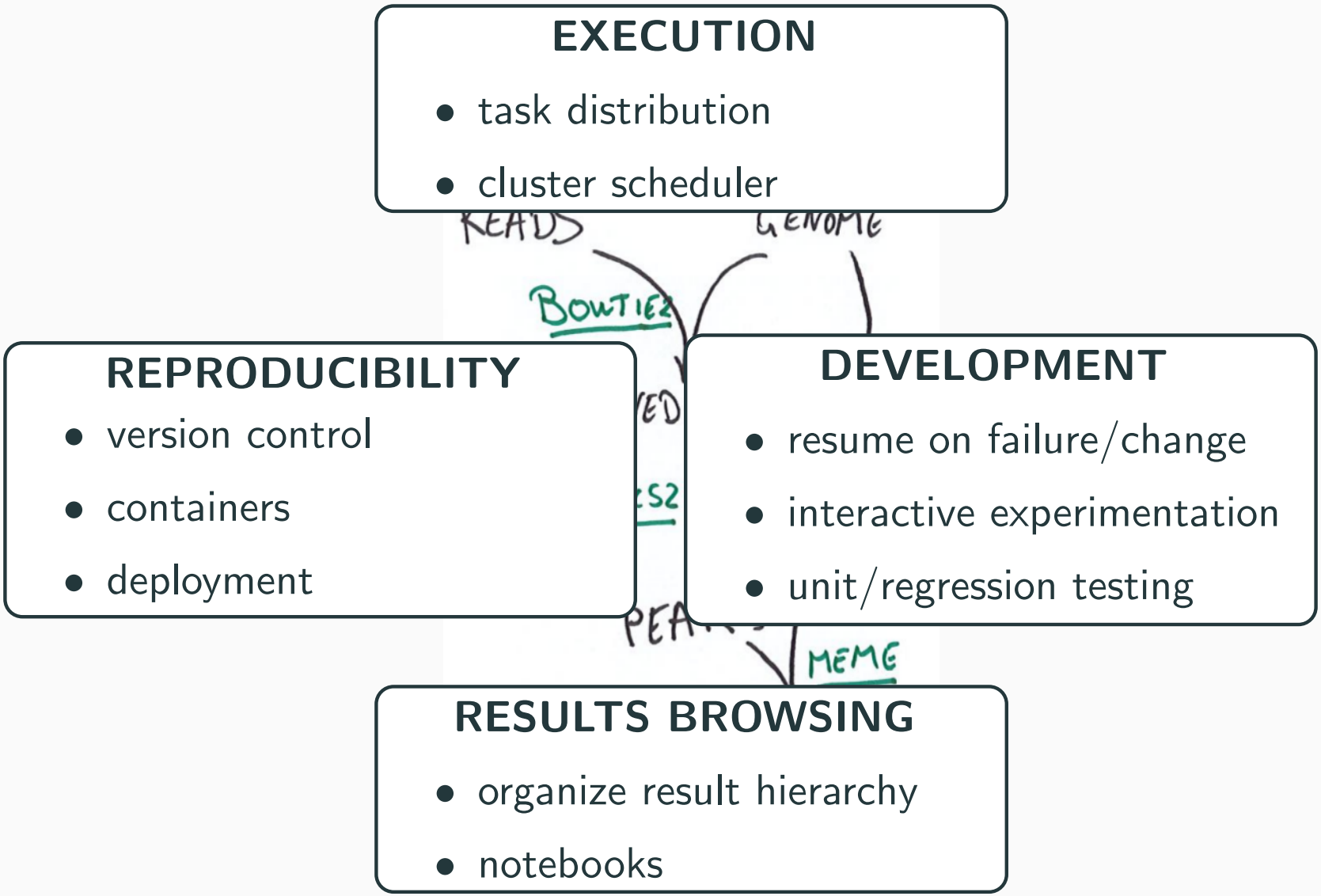
REPRODUCIBILITY

- version control
- containers
- deployment

DEVELOPMENT

- resume on failure/change
- interactive experimentation
- unit/regression testing

Motivations



Pipelines in computational biology

Typically

- from a few to 10k or 100k steps
- each step is a call to a independant program/script
- many "plumbing" scripts between analysis programs

`bash` is a popular, but terrible option :

- Programming errors discovered only when commands are executed
  - syntax errors
  - typos on commands, paths, options
  - use of inappropriate file formats
- Resuming after an error or a modification is awkward
- Naming and managing intermediate files is a pain
- No simple way to distribute calculations
- No way to make sure that result files are up-to-date

A tough situation for scientists...

Pipelines are complex pieces of software

- many programs, usually in many different languages
- no tools to ensure the plumbing between them is correct

Developing a reproducible scientific pipeline is excruciatingly difficult

- requires inhuman attention
- and skills using very varied computer science tools

All of this diverts us from the actual data analysis

Are our best practices really helping?

Proposal

Scientific pipelines are complex pieces of software

☞ We need to apply good old software engineering recipes!

- Code reuse

  Don't apply best practices, *implement them* once and for all in a *reusable library*

- Separation of concern

  Declarative pipeline construction independent of its execution

- Abstraction

  Hide a maximum of technical details, provide a uniform API

- Composition

  use simple notions like functions and typing to create arbitrarily complex pipelines easily

↪ implemented in an OCaml library: **bistro**

First steps with Bistro

Overview

`bistro` is an OCaml library consisting of three main components:

1. a module `Bistro` that introduces a type `α workflow` representing a computation
2. a module `Bistro_engine` that implements a scheduler to actually run the recipes
3. a module `Bistro_bioinfo` that provides workflow constructors for many standard tools in computational biology

The `α workflow` type

- `α workflow` represents a set of interdependent computational steps that produce a single result of type α
- each step can be described as a shell script or an OCaml function
- this script will typically refer to other workflows, which are **dependencies** of the workflow being defined
- the result of a workflow is typically cached "somewhere"
  ☞ in a location which depends on the code of the script/function

OCaml syntax

An OCaml program is a sequence of definitions:

```
let i = 0;;          (* define an integer variable named [i] *)
let j = i + 1;;      (* reuse a previous definition to make a new one *)
let s = "bistro";;   (* a string variable *)
```

For functions, the syntax is:

```
let f x = x + 1;;    (* function definition *)
let k = f i;;        (* function call *)
let g x y = x + y;;  (* function with several arguments *)
let l = g i j;;      (* calling a function with several arguments *)
```

Programs are typically fed to:

- an interpreter (like in `python` or `R`)
- or a compiler to produce an executable

Typing

Every expression has a type, which can be inferred **before** execution:

```
# let i = 0;;
val i : int = 0

# let j = i + 1;;
val j : int = 1

# let s = "bistro";;
val s : string = "bistro"

# let f x = x + 1;;
val f : int -> int = <fun>

# let g x y = x + y;;
val g : int -> int -> int = <fun>
```

Parameterized types

Types can have a parameter that expresses additional details on values:

```
# let l1 = [ 1 ; 2 ; 3 ];;          (* a list of integers *)
val l1 : int list = [1; 2; 3]

# let l2 = [ "a" ; "b" ; "c" ];; (* a list of strings *)
val l2 : string list = ["a"; "b"; "c"]

# let l3 = [];;                          (* an empty list *)
val l3 : α  list = []
```

You do not mess with the compiler

Types are inferred and checked to detect programming errors

```
# let f x = x + 1;;
val f : int -> int = <fun>

# f [ 1 ; 2 ];;
    ^^^^^^^^^
Error: This expression has type α  list
       but an expression was expected of type int
```

Our first workflows

```
# let i = Workflow.int 41;; (* A constant integer workflow *)
val i : int workflow = <abstr>

(* This is one way to add a step to a preexisting workflow x *)
# let%workflow f x =
    [%eval x] + 1;;
val f : int workflow -> int workflow = <fun>

(* More complex workflows are simply built by function application *)
# let answer = f i;;
val answer : int workflow = <abstr>
```

`[%eval ...]` is used to access the result of a workflow in the definition of another workflow.

Path workflows (1)

`α path` is an abstract type representing paths in the filesystem. It is typed to represent the format of the file.

```
# let data = Workflow.input "data.tsv";; (* Input file workflow *)
val data : α path workflow = <abstr>

# let%workflow wc file =      (* using path workflows *)
    In_channel.read_lines [%path file]
    |> List.length;;
val wc : α  path workflow -> int workflow = <fun>
```

`[%path ...]` is used to access the location where the result of a path workflow is stored.

Path workflows (2)

```
(* Definition of a path workflow *)
# let%pworkflow remove_comments file =
    In_channel.read_lines [%path file]
    |> List.filter ~f:(Fn.not (String.is_prefix ~prefix:"#"))
    |> Out_channel.write_lines [%dest];;
val remove_comments : α path workflow -> β path workflow = <fun>

# let nb_points = wc (remove_comments data);;
val nb_points : int workflow = <abstr>
```

[%dest] represents the location where to save the result of a path workflow.

Shell workflows

```
# let wget url = Workflow.bash [%script "
    # Here I can write a bash script
    wget -O {{dest}} {{string url}}
  "];;
val wget : string -> α path workflow = <fun>
```

- Python, R, perl scripts can be created the same way
- other shell workflow constructors are available for more complex wrapping

Tools available in `Bistro_bioinfo`

The list is regularly expanding, currently :

- Art
- Bed
- Bedtools
- Bowtie2
- Bowtie
- ChIPQC
- Deeptools
- Deseq2
- Diamond
- Ea_utils
- Ensembl
- Fastool
- FastQC
- Fastq
- Fastq_screen
- Hisat2
- Htseq
- Idba
- Kallisto
- Macs2
- Meme_suite
- Picard_tools
- Prokka
- Quast
- Samtools
- Silix
- Spades
- Sra_toolkit
- Srst2
- Transdecoder
- Tophat
- Trinity
- Ucsc_gb
- Velvet

A high-level interface

OCaml module interfaces offer a powerful way to build clear and highly reusable APIs (here on Unix tools)

```
module Bistro_unix : sig
  val wget :
    ?no_check_certificate:bool ->
    ?user:string ->
    ?password:string ->
    string -> #file pworkflow

  val gunzip : α gz pworkflow -> α pworkflow
  val bunzip2 : α bz2 pworkflow -> α pworkflow

  val head :
    n:int ->
    #text_file pworkflow ->
    #text_file pworkflow
end
```

Another example

The type of wrappers summarizes each tool's interface and is used by the compiler to check our pipeline

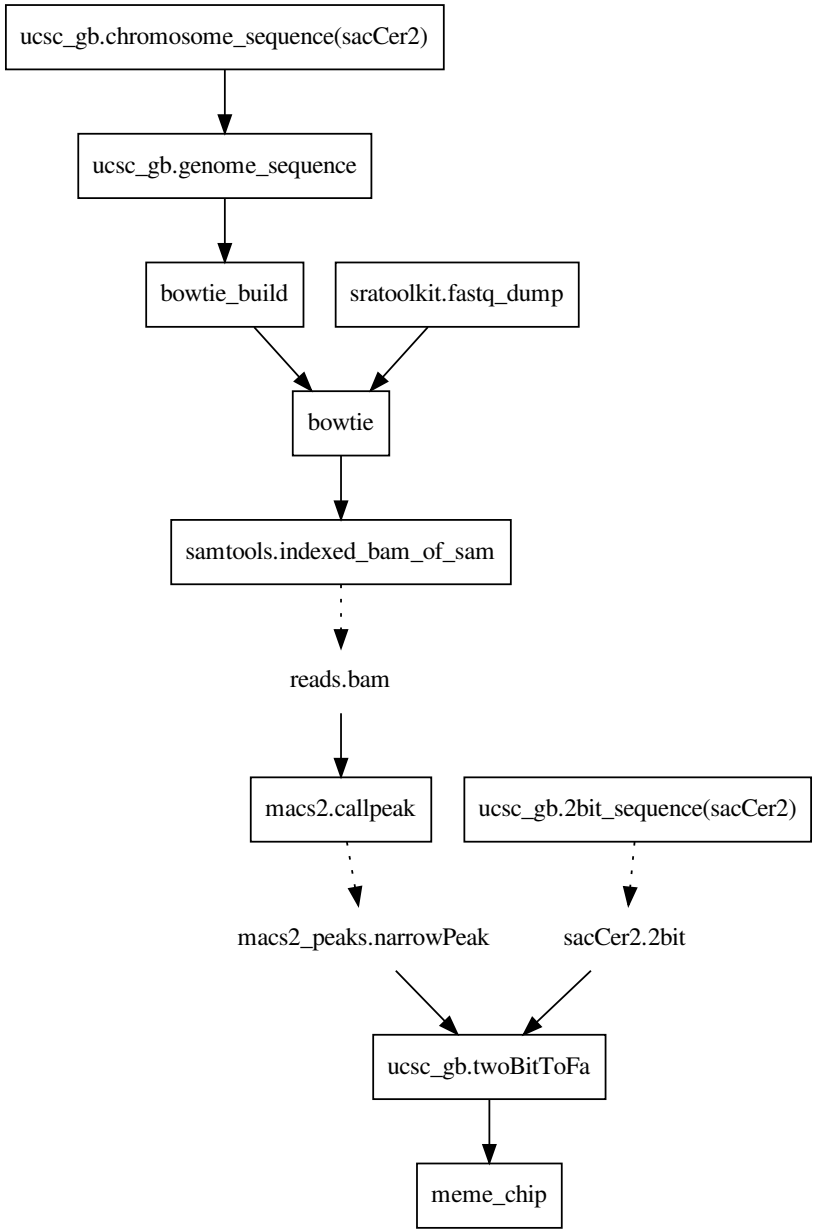
```
module Bowtie2 : sig
  val bowtie2_build :
    ?large_index:bool ->
    ?noauto:bool ->
    ?packed:bool ->
    ?bmax:int ->
    ?bmaxdivn:int ->
    ?dcv:int ->
    (* [...] *)
    ?seed:int ->
    ?cutoff:int ->
    fasta pworkflow ->
    index pworkflow
end
```

A typical (small) pipeline

```
open Bistro_bioinfo
open Bistro_utils

let sample_fq = Sra_toolkit.fastq_dump (`id "SRR217304")
let genome = Ucsc_gb.genome_sequence `sacCer2
let index = Bowtie.bowtie_build genome
let mapped_reads =
  Bowtie.bowtie ~v:2 index (`single_end [sample_fq])
let peaks =
  Macs2.(callpeak ~qvalue:100. sam [ mapped_reads ] / narrow_peaks);;
let genome_2bit = Ucsc_gb.genome_2bit_sequence `sacCer2
let sequences = Ucsc_gb.twoBitToFa peaks genome_2bit
let motifs = Meme_suite.meme_chip sequences

let repo = Repo.[
  item [ "peaks" ] peaks ;
  item [ "motifs" ] motifs
]

let loggers = [ Console_logger.create () ]
let () = Repo.build ~loggers ~np:4 ~mem:(`GB 4) ~outdir:"res" repo
```

A typical (small) pipeline

```
open Bistro_bioinfo
open Bistro_utils

let sample_fq = Sra_toolkit.fastq_dum
let genome = Ucsc_gb.genome_sequence
let index = Bowtie.bowtie_build genom
let mapped_reads =
  Bowtie.bowtie ~v:2 index (`single_e
let peaks =
  Macs2.(callpeak ~qvalue:100. sam [
let genome_2bit = Ucsc_gb.genome_2bit
let sequences = Ucsc_gb.twoBitToFa pe
let motifs = Meme_suite.meme_chip seq

let repo = Repo.[
  item [ "peaks" ] peaks ;
  item [ "motifs" ] motifs
]

let loggers = [ Console_logger.create
let () = Repo.build ~loggers ~np:4 ~m
```



Executing a workflow

- up to now, we have just described a pipeline, nothing was run
- define output files of the analysis, and the way they should be organized in a directory

```
let repo = Repo.[
  item [ "peaks" ] peaks ;
  item [ "motifs" ] motifs ;
];;
```

- actually run the pipeline specifying resources and (optional) logging

```
let loggers = [ Console_logger.create () ];;    (* Logs event on standard output *)
let () = Repo.build ~loggers ~np:4 ~mem:(`GB 4) ~outdir:"res" repo;;
```

- this will create a result directory equivalent to

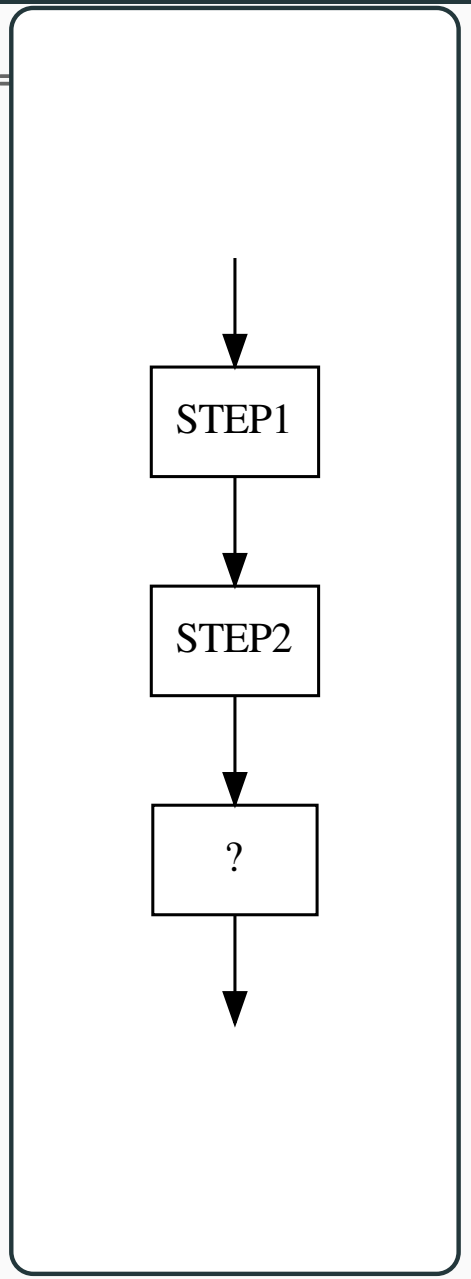
```
res
|-- motifs
|   |-- index.html
|   |-- meme_out
|   `  [...]
`-- peaks
```

Complex, generic pipelines: just use functions!

```
let one_sample_analysis mapping_meth s =
  sample_data s
  |> step1 ~param:true
  |> step2
  |> mapping_meth

let pipeline mapping_meth samples =
  List.map one_sample_analysis samples
  |> differential_analysis

let comparison_pipeline samples =
  compare_results
    (pipeline mapping_meth1 samples)
    (pipeline mapping_meth2 samples)
```

Complex, generic pipelines: just use functions!

```
let one_sample_analysis mapping_meth s =
  sample_data s
  |> step1 ~param:true
  |> step2
  |> mapping_meth

let pipeline mapping_meth samples =
  List.map one_sample_analysis samples
  |> differential_analysis

let comparison_pipeline samples =
  compare_results
    (pipeline mapping_meth1 samples)
    (pipeline mapping_meth2 samples)
```

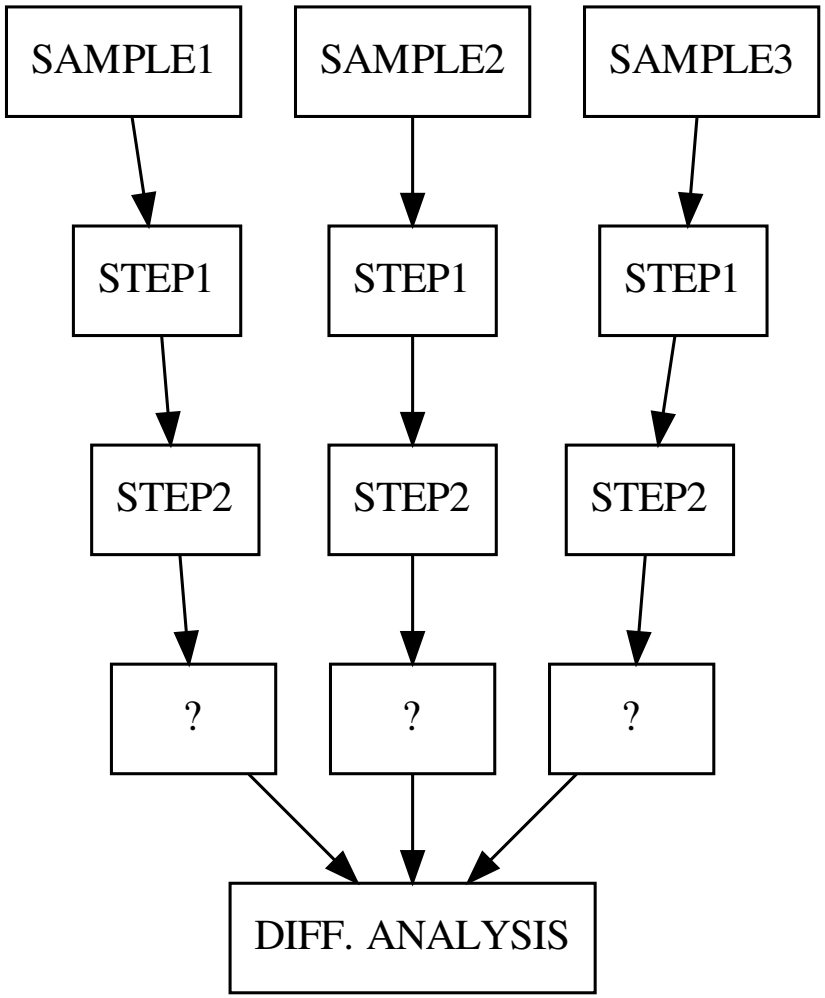


Complex, generic pipelines: just use functions!

```
let one_sample_analysis mapping_meth s =
  sample_data s
  |> step1 ~param:true
  |> step2
  |> mapping_meth

let pipeline mapping_meth samples =
  List.map one_sample_analysis samples
  |> differential_analysis

let comparison_pipeline samples =
  compare_results
    (pipeline mapping_meth1 samples)
    (pipeline mapping_meth2 samples)
```

Complex, generic pipelines: just use functions!

```
let one_sample_analys
  sample_data s
  |> step1 ~param:tru
  |> step2
  |> mapping_meth

let pipeline mapping_
  List.map one_sample
  |> differential_ana

let comparison_pipeli
  compare_results
    (pipeline mapping
    (pipeline mapping
```

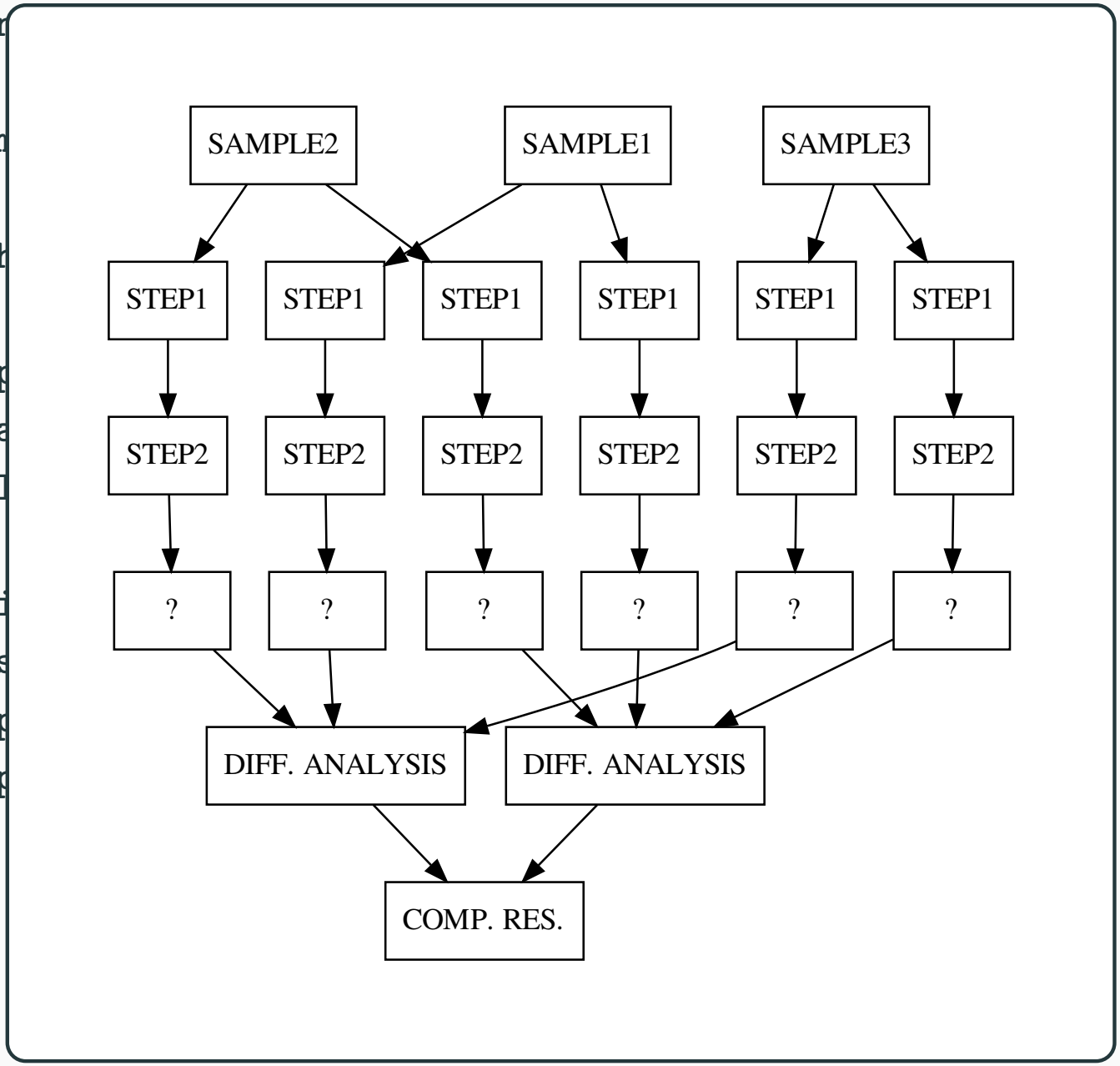


Complex, generic pipelines: just use functions!

```
let one_sample_analysis mapping_meth s =
  sample_data s
  |> step1 ~param:true
  |> step2
  |> mapping_meth

let pipeline mapping_meth samples =
  List.map one_sample_analysis samples
  |> differential_analysis

let comparison_pipeline samples =
  compare_results
    (pipeline mapping_meth1 samples)
    (pipeline mapping_meth2 samples)
```

Complex, generic pipelines: just use functions!

```
let one_sample_ar
  sample_data s
  |> step1 ~paran
  |> step2
  |> mapping_meth

let pipeline mapp
  List.map one_sa
  |> differential

let comparison_pi
  compare_results
    (pipeline map
    (pipeline map
```



What did we gain?

Distributed execution

- use task independance to run as many commands as possible simultaneously
- each task may be given several processors
- control over available number of processors and total memory
- if required, intermediate files are deleted when they are not needed anymore

Resume-on-failure, resume-on-change

- if some step fails, correct it and run again
- the scheduler will start from where it stopped automatically
- only needed tasks will be run again
- same thing when modifying the pipeline during development

As an example, after changing

```
let wget url =
  Workflow.bash [%script "wget -O {{dest}} {{string url}}"]
```

to

```
let wget url =
  Workflow.bash [%script "wget -F -O {{dest}} {{string url}}"]
```

all workflows built with `wget` and those that depend on them will be rebuilt automatically.

Painless deployment

- only required install: OCaml + bistro
- easy and portable thanks to OPAM (OCaml package manager)
- all tools will be downloaded on the fly
- with the exact version specified in bistro
- no actual install on the system

- this is achived using Docker or Singularity containers
- can be turned off (and then bistro assumes tools are installed on the system)

Logging/reporting

Console output for events

```
[2017-09-30 20:04:52.000000+02:00] started ucsc_gb.2bit_sequence(sacCer2).fd7a33
[2017-09-30 20:04:52.000000+02:00] started sra.fetch_srr(SRR217304).8d256e
[2017-09-30 20:04:52.000000+02:00] started ucsc_gb.chromosome_sequences(sacCer2).20c330
[2017-09-30 20:04:52.000000+02:00] started ucsc_gb.fetchChromSizes.ea1967
[2017-09-30 20:05:10.000000+02:00] ended ucsc_gb.fetchChromSizes.ea1967 (success)
[2017-09-30 20:05:10.000000+02:00] ended ucsc_gb.2bit_sequence(sacCer2).fd7a33 (success)
[2017-09-30 20:05:42.000000+02:00] ended sra.fetch_srr(SRR217304).8d256e (success)
[2017-09-30 20:05:42.000000+02:00] started sratoolkit.fastq_dump.932827
[...]
```

Logging/reporting

Console output for errors

```
################################################################################
#                                                                              #
#  Task a0ef08ae3b09f1dc80b6cf9c2aa6a5c2 failed
#
#------------------------------------------------------------------------------#
#
# Ended with exit code 255
#                                                                              #
################################################################################
###
##
#
+------------------------------------------------------------------------------+
| Submitted script                                                             |
+------------------------------------------------------------------------------+
(docker run --log-driver=none --rm -v /home/pveber/w/2017-10-02-groupe-ngs/code/_bistro/cache/fd7a337e1fc261da9e387a17c00e7b8b/
#
+------------------------------------------------------------------------------+
| STDOUT                                                                       |
+------------------------------------------------------------------------------+

+------------------------------------------------------------------------------+
| STDERR                                                                       |
+------------------------------------------------------------------------------+
twoBitReadSeqFrag in chrI end (230319) >= seqSize (230208)
```

Logging/reporting

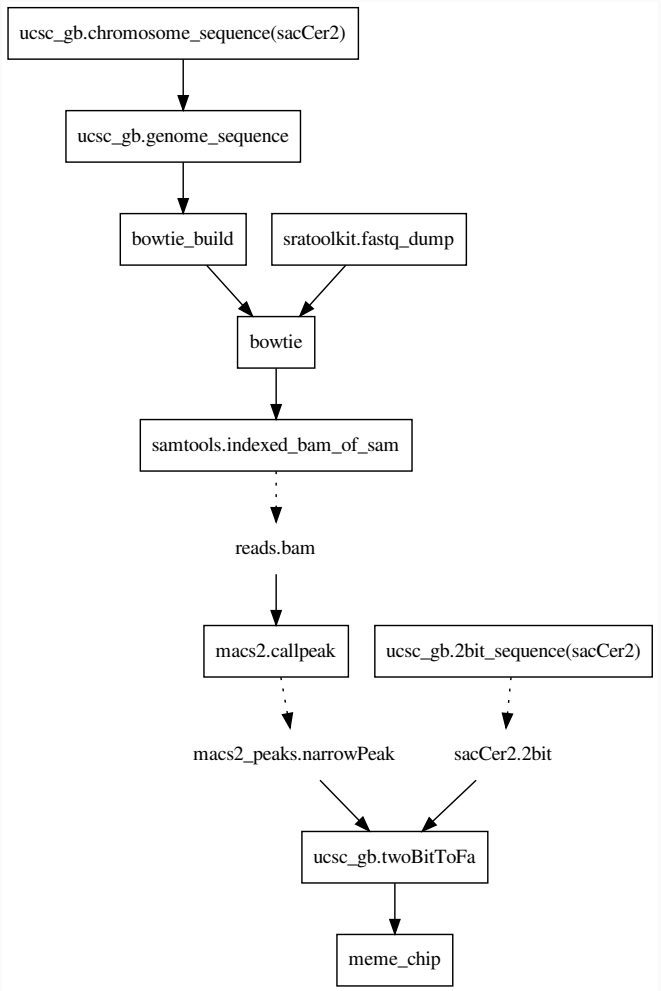
HTML execution report

EVENT LOG

2017-09-30 20:18:14	**STARTED**	macs2.callpeak
2017-09-30 20:18:14	**DONE**	bowtie **id:** 3ef1d878c01223b44b99352b2f6835ab **outcome:** stdout stderr **command:** `(docker run --log-driver=none --rm -v /home/pveber/w/2017-10-02-groupe-ngs/code/_bistro/cache/93282712a92c50c18e682ffcdf94b1a7:/bistro/data/9de893ff548ae2d8390ae859fa4349d7 -v /home/pveber/w/2017-10-02-groupe-ngs/code/_bistro/cache/bcae1f847b9ce3f5ca63499c3ae261f1:/bistro/data/58e996b4443fe25ab52e4bb3c09504cf -v /home/pveber/w/2017-10-02-groupe-ngs/code/_bistro/tmp/3ef1d878c01223b44b99352b2f6835ab/tmp:/bistro/tmp -v /home/pveber/w/2017-10-02-groupe-ngs/code/_bistro/tmp/3ef1d878c01223b44b99352b2f6835ab:/bistro -i pveber/bowtie:1.1.2 bash -c 'bowtie -S -v 2 -p 4 /bistro/data/58e996b4443fe25ab52e4bb3c09504cf/index /bistro/data/9de893ff548ae2d8390ae859fa4349d7/bistro/dest')`
2017-09-30 20:17:45	**STARTED**	bowtie
2017-09-30 20:17:45	**DONE**	bowtie_build
2017-09-30 20:17:45	**DONE**	sratoolkit.fastq_dump
2017-09-30 20:17:13	**STARTED**	bowtie_build

Logging/reporting

Task graph representation



Petty error detector

Compiler assistance **before running the pipeline**

- against syntax errors

```
# let mapped_reads = Bowtie.bowtie ~v:2 index (`single_end [sample_fq];;
                                                ^

Error: Syntax error: ')' expected, the highlighted '(' might be unmatched
```

- against typos

```
# let index = Bowtie.boqtie_build genome;;
              ^^^^^^^^^^^^^^^^^^

Error: Unbound value Bowtie.boqtie_build
Hint: Did you mean bowtie_build?
```

- against inappropriate formats

```
# let index = Bowtie.bowtie_build sample_fq;;
                                  ^^^^^^^^^

Error: This expression has type sanger_fastq pworkflow
but an expression was expected of type fasta pworkflow
```

Significant reduction of mental load

- no need to find names for intermediate files, nor to care about them at all
- by construction, impossible to give a tool a wrong path
- no need to remember how programs should be called
- the type of OCaml functions can be followed to remember how to use a tool (with assistance from the compiler)

More time to think on the pipeline steps!

Final words

Many benefits of a library embedding :

- a LOT more code reuse between projects
- advanced workflow construction
  - map sample collections, optional parts
  - use of functors to enhance pipeline reuse
- **Costless derivation of web interfaces for workflows**

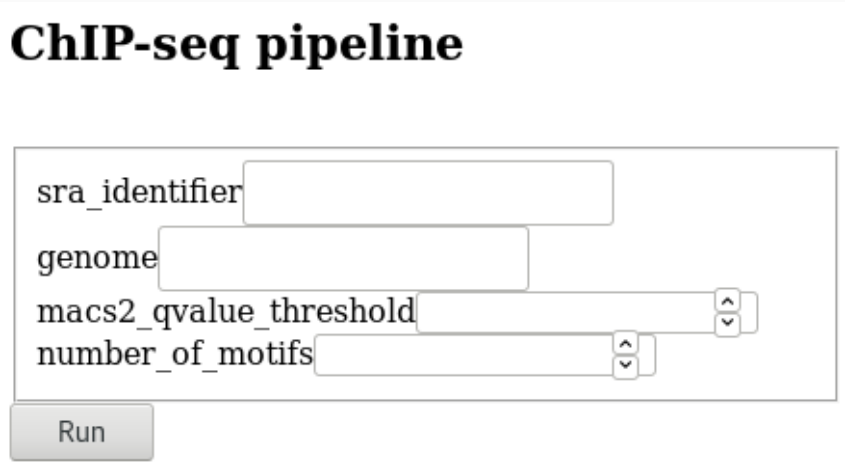
Current developments:

- Multi-node distribution
- Notebook publication system

Web interface for workflows (preview)

Define an input to your pipeline, and automatically derive an input form

```
type input = {
  sra_identifier : string ;
  genome : string ;
  macs2_qvalue_threshold : float ;
  number_of_motifs : int ;
}
[@@deriving bistro_form]
```



Web interface for workflows (preview)

```
module ChIP_seq_pipeline = struct
  type input = {
    sra_identifier : string ;
    genome : string ;
    macs2_qvalue_threshold : float ;
    number_of_motifs : int ;
  }
  [@@deriving sexp]

  let title = "ChIP-seq pipeline"
```

Web interface for workflows (preview)

```
let derive ~data i =
  let sample_sra = Sra.fetch_srr i.sra_identifier in
  let sample_fq = Sra_toolkit.fastq_dump sample_sra in
  let org = genome_of_string i.genome in
  let genome = Ucsc_gb.genome_sequence org in
  let index = Bowtie.bowtie_build genome in
  let mapped_reads =
    Bowtie.bowtie ~v:2 index (`single_end [sample_fq]) in
  let peaks =
    Macs2.(callpeak ~extsize:150 ~nomodel:true
             ~qvalue:i.macs2_qvalue_threshold
             sam [ mapped_reads ] / narrow_peaks)
  in
  let genome_2bit = Ucsc_gb.genome_2bit_sequence org in
  let sequences =
    Ucsc_gb.twoBitToFa
      Ucsc_gb.(bedClip (fetchChromSizes `sacCer2) (Bed.keep4 peaks))
      genome_2bit
  in
  let motifs = Meme_suite.meme_chip ~meme_nmotifs:i.number_of_motifs sequences in
  Bistro_repo.[
    [ "QC" ] %> FastQC.run sample_fq ;
    [ "peaks" ] %> peaks ;
    [ "motifs" ] %> motifs
  ]
end
```

Web interface for workflows (preview)

```
module Server = Bistro_server.Make(ChIP_seq_pipeline)

let () = Server.start ()
```

In the end, 60 lines for a web server providing a basic ChIP-seq analysis service...