

# Table of contents

|                                                                    |          |
|--------------------------------------------------------------------|----------|
| <b>Using LLMs to understand and check code</b>                     | <b>2</b> |
| Why learn the command line if an LLM can write the code? . . . . . | 2        |
| Start with the basics, without an LLM . . . . .                    | 3        |
| Which tool can you use? . . . . .                                  | 4        |
| Rules at the UvA . . . . .                                         | 4        |
| Different tools, same advice . . . . .                             | 4        |
| How I suggest you use an LLM . . . . .                             | 5        |
| Using an LLM to understand code . . . . .                          | 11       |
| Checking code with the commands from day 1 . . . . .               | 14       |
| Worked example: an LLM writes a fastp job script . . . . .         | 16       |
| Less code is often better . . . . .                                | 21       |
| When your code does not work . . . . .                             | 25       |
| Further reading . . . . .                                          | 26       |

# Using LLMs to understand and check code

Tools like ChatGPT, Claude, Copilot or Gemini can write bash code in seconds, so why would you still want to learn how to use bash?

For this, let's first understand what these tools are. Very briefly, all these tools are built on a **large language model (LLM)**: a program that has been trained on huge amounts of text, including a lot of code, and that writes an answer by predicting which words are likely to come next. On this page, I use "LLM" for all of these chat tools.

LLMs are useful, and some of you may already use them. This page is about how to use them without losing control over your analysis. It is written based on my own experience using LLMs, for people who work with biological data and who have just started with the command line. You can read it after the workshop, at your own pace.

The short version:

- **Learn the basics first**, with this tutorial, and don't let an LLM write code for you while you do. The basics give you the tools you need for all the other advice on this page
- Both while you learn and later, use an LLM as a **tutor** that explains code, more than as a machine that writes code for you
- Treat every piece of code an LLM writes as a **first draft**: run it on one sample and check the output before you use it for real
- You need to be able to **explain** and **justify** every step of your analysis, whoever wrote the code
- You need to **understand** what each option of a tool does to your data. For the code around it, you can **check** that it does what you want with the simple commands from day 1

## Why learn the command line if an LLM can write the code?

This is a fair question. In my opinion, even if you use an LLM, someone still has to:

- **Read the code**, to know what it will do to your files before you run it. This means understanding details like how exactly the data is summarized or how the options of a command, for example `fastp`, affect your output
- **Run the code** in the right place: on the right computer, in the right folder, with the right paths, on the login node or as a job. The LLM does not know your folder structure, your file names or the HPC you work on
- **Check the input and output**: what does your input data look like, did every sample get processed, how many reads are left, does that make sense for your data?
- **Explain and justify** your choices: to your supervisor, to your co-authors, in the methods section of a paper. “The LLM chose this cutoff” is not really an acceptable answer if someone asks you about your choices

All four need the skills from this workshop. An LLM can help with each of them, but it cannot take over the responsibility. That stays with you.

## Start with the basics, without an LLM

All the advice on this page depends on the basics from this tutorial. To check code you need to be able to navigate the file system with the terminal and know about basic commands like `ls`, `wc -l` or `head`. To notice that a loop skipped a sample, you need to have written a few loops yourself. And to see that an output looks wrong, you need to know what a normal output looks like.

So while you learn the basics, with this tutorial and your first own analyses, my advice is:

- **Don’t use an LLM to write code**. Type the commands yourself, make mistakes and read the error messages. This is how you learn what normal looks like, and you need exactly that later, to spot when something is wrong. Struggle is a normal part of the learning experience and you don’t want to outsource this to an LLM.
- **Use an LLM only when you don’t understand something and can’t ask anyone**: not your supervisor, not a colleague, not the instructor. Even then, ask it to explain, not to give you the solution

Once you are comfortable with the basics, the rest of this page applies.

## Which tool can you use?

### Rules at the UvA

Universities have their own rules for LLMs, mostly about privacy. At the UvA (checked in October 2026, rules may change):

- The UvA offers its own tool, [UvA AI Chat](#), to students and staff. It is free with your UvAnetID and works much like ChatGPT. According to UvA, what you type stays within the UvA environment and is not used to train the models
- For **students**, UvA writes that commercial tools such as ChatGPT are [not allowed in the context of your studies](#), because they don't meet the UvA privacy rules. Your study programme and your course decide what you may use an LLM for, and you are expected to be open about how you used it
- For **everyone**, also in UvA AI Chat, the normal rules for privacy and research data still apply. The UvA's [page on responsible use](#) says, for example, never to upload lists of names of students, patients or research participants, and to contact your faculty's data steward when in doubt

If you are not at the UvA, check the rules of your own institution.

**!** Important: Don't paste your data into an LLM

Don't share confidential information, unpublished data, sequences or results with an LLM. You rarely need to. To ask about a problem, a few lines are enough: the command you ran, the error message, and the first lines of a file (`head -n 8` shows the first two reads of a fastq file). Replace sample names or other details if they are sensitive.

### Different tools, same advice

LLM tools differ a lot: free or paid, a chat window in your browser, or a tool inside a code editor that can read your files and run commands itself. Names, features and prices change every few months, so I don't recommend a specific one here.

Everything on this page works with a free chat tool, such as UvA AI Chat: you paste code, you ask a question, you read the answer. If you use a tool that runs commands for you, the advice matters even more. You see less of what happens, so checking the result becomes even more important.

## How I suggest you use an LLM

These are the rules I give the students I supervise for how to interact with an LLM. As I mentioned above, I advise students to not let an LLM write their code and not outsource their learning. Your internships are not only for you to produce results, they give you the time and space to learn about the process of producing good and reliable results. However, I am also aware that everyone has access to these tools, so these recommendations also cover how I would use LLMs for coding:

- **Ask it to explain, not only to write.** When you learn a new concept, you can even tell it not to write code, for example: “Give advice and best practices, but do NOT give me a code solution”
- **Ask specific questions.** Give it the actual command, the actual error message and the biological context. The more specific the question, the more useful the answer
- **Ask for the simplest version.** LLMs tend to write more code than you need: error handling for every possible case, a variable for every path. The code is not always wrong, but it is harder to understand what it does. Ask for the simplest version first, and add complexity only when you understand why you need it
- **Treat generated code as a first draft.** Run it on one sample, look at the output, and check that it makes biological sense before you run it on all samples
- **Don’t let it rewrite code that fails.** This usually replaces code you don’t understand with more code you don’t understand. Ask it to explain the error or a specific line instead (more on this [below](#))
- **Check suggested options against the help page of the version you use.** An LLM has learned from documentation of many versions of a tool, often older ones. Remember from day 2: Crunchomics has fastp 1.0.1, and we installed 1.4.0. Options can change between versions. `fastp --help` on your own installation is always more reliable than an LLM’s suggestion
- **Write down what each result means, in your own words.** Your notebook is not only a record of what you did. It is also a place to slow down and think. After each analysis step, write at least one sentence about what the result means for your data, for example: “fastp removed 3% of the reads” and also think about whether that number is something that you expected. Make sure to write this yourself, not with an LLM

And the most important rule: **if you can’t explain a step, don’t use it yet.** It is completely fine to ask for help with a step you don’t understand, from a person or from an LLM. Struggling with code is a normal part of learning.

## Exercise

On day 2, we ran `fastp` without any filtering options. Let's use an LLM to find out whether it filtered all reads anyway:

1. Ask in UvA AI Chat (or another chat tool that you are allowed to use): "What does `fastp` do by default to filter reads? Which options control this, and what are their default values?" Write down its answer
2. On Crunchomics, activate the `fastp_1.4.0` environment and look at the help page. Find the options the LLM mentioned. Do the descriptions and default values match?
3. Deactivate the environment and look at the help page of the `fastp 1.0.1` of Crunchomics. Is there anything different?

**Click me to see an answer**

**The help page of `fastp 1.4.0` (question 2)**

```
# Question 2: the help page of fastp 1.4.0
conda activate fastp_1.4.0
fastp --version
fastp --help 2>&1 | less

# Question 3: the help page of the fastp of Crunchomics
conda deactivate
fastp --version
fastp --help 2>&1 | less
```

`2>&1` sends the error output to the same place as the normal output. Some tools print their help page as error output, and then `less` would not get it. Inside `less`, type `/quality` and press Enter to jump to the word "quality", and `n` to jump to the next match.

Look for the options about quality and length filtering. For `fastp 1.4.0` on Crunchomics, these are the lines:

```
-Q, --disable_quality_filtering      quality filtering is
enabled by default. If this option is specified, quality
filtering is disabled
-q, --qualified_quality_phred        the quality value that a
base is qualified. Default 15 means phred quality >=Q15 is
qualified. (int [=15])
```

```

-u, --unqualified_percent_limit      how many percents of bases
are allowed to be unqualified (0~100). Default 40 means 40%
(int [=40])
-n, --n_base_limit                   if one read's number of N
base is >n_base_limit, then this read/pair is discarded.
Default is 5 (int [=5])
-e, --average_qual                   if one read's average
quality score <avg_qual, then this read/pair is discarded.
Default 0 means no requirement (int [=0])
-L, --disable_length_filtering       length filtering is
enabled by default. If this option is specified, length
filtering is disabled
-l, --length_required                reads shorter than
length_required will be discarded, default is 15. (int [=15])

```

In plain words:

- **-q** and **-u**: a base with a quality score of 15 or more counts as good. A read is removed if more than 40% of its bases are not good
- **-n**: a read is removed if it has more than 5 N bases (positions where the sequencer could not tell which base it was)
- **-e**: the minimum average quality of a read. The default is 0, so this filter is off unless you switch it on
- **-l**: a read is removed if it is shorter than 15 bases

So yes: even without options, fastp filtered the reads on day 2. Your fastp report from day 2 shows how many reads were removed, and by which filter.

Filtering is not the only thing that is on by default. The same help page says that adapter trimming is on by default (**-A** switches it off), while quality trimming with **--cut\_front**, **--cut\_tail** or **--cut\_right** is off until you ask for it. So “fastp trims and filters my reads” is only half true, and which half depends on the option. This decides what “I ran fastp with default settings” actually means, for example in the methods section of a paper.

### **What two LLMs answered (question 1, October 2026)**

I asked this question to UvA AI Chat (with its default model) and to ChatGPT, and compared both answers with the help page of fastp 1.4.0. I used ChatGPT on purpose, without any memory of earlier chats, because it knows nothing about me. UvA AI Chat does remember my earlier

conversations (see point 5 below). As a student, use UvA AI Chat (see the UvA rules above).

1. Both got every default value right, but they also didn't give me the full output of the help function. Checking this took me a few minutes, and that is the point: often the LLM is right and often it might only give you a partial answer based on what you asked. A quick look at the help page is how you know
2. Quality trimming was hidden in one answer, in a side note near the end, and missing from the other. Both answered the question I asked, which was about filtering. But for most of us, another relevant question is "were my reads trimmed?". With Illumina data, the quality often drops towards the end of the read, so you may want to cut those ends off. To ask about that, though, you first need to know it. As I mentioned in point 1, an LLM answers only what you ask. It may tell you only part of what matters for your data, and nothing in the answer tells you that something is missing. The help page doesn't depend on your question: it lists every option, including the ones that are off and the ones you didn't know to ask about.
3. Both wrote that fastp removes reads shorter than 15 bases "after trimming". That may well be true, but the help page doesn't say it. One answer also stayed vague ("fastp *can* discard a pair . . . depending on . . .") about what happens to the other read of a pair when one read fails, while the help page says clearly that it is discarded by default
4. Neither answer said which version of fastp it described
5. One answer ended with advice tailored to one of my own research projects, which I had not mentioned in the question. It assumed a different kind of data than the data I was asking about. The reason for that is that depending on your settings an LLM can remember earlier conversations. This can be useful for some things but you have to decide if this is what you want and whether this fits into the privacy rules of your organisation.

### Differences between versions (question 3)

Scrolling through two long help pages is tiring. Instead, we can save both in a file and let the computer compare them. `diff` compares two files line by line and prints only the lines that differ:

```
# Save the help page of each version in a file
conda activate fastp_1.4.0
fastp --help > help_1.4.0.txt 2>&1
conda deactivate
fastp --help > help_1.0.1.txt 2>&1

# Print only the lines that differ
diff help_1.0.1.txt help_1.4.0.txt
```

On Crunchomics, this prints:

```
16c16
<      --stdin                      input from STDIN. If the
STDIN is interleaved paired-end FASTQ, please also add
--interleaved_in.
---
>      --stdin                      input from STDIN. If the
STDIN is interleaved paired-end FASTQ, please also add
--interleaved_in. Adapter auto-detection is disabled for STDIN
mode
28a29
>      --dimer_max_len              if the read length is
less than or equal to this value after adapter trimming, it is
considered an adapter dimer. Requires adapter evidence. (int
[=2])
36c37
<      --dup_calc_accuracy          accuracy level to
calculate duplication (1-6), higher level uses more memory (1G,
2G, 4G, 8G, 16G, 24G). Default 1 for no-dedup mode, and 3 for
dedup mode. (int [=0])
---
>      --dup_calc_accuracy          accuracy level to
calculate duplication (1-6), higher level uses more memory (1G,
2G, 4G, 8G, 16G, 32G). Default 1 for no-dedup mode, and 3 for
dedup mode. (int [=0])
```

Lines that start with < come from the first file (1.0.1), lines with > from the second file (1.4.0). The numbers above them say where in each file the lines are.

So for the filtering options, nothing changed between 1.0.1 and 1.4.0. That

is a good outcome too: the point of the check is to *know*, not to find a difference. Three other lines did change. The most interesting one is `--dimer_max_len`, a new option for adapter trimming in 1.4.0. Reads of 2 bases or less would be removed by the length filter anyway, but the help page doesn't say whether the new option changes anything else. Interestingly, the fastp README on GitHub for version 1.4.0 doesn't list this option at all (checked in October 2026). So even the official documentation online can lag behind the program.

### Key points

- Often the LLM is right but it might also blur things in its response. A quick look at the help page is how you *know*
- An LLM answers only what you ask, and it won't tell you what you didn't ask. The help page lists every option, also the ones that are off
- "Default settings" only means something if you know what is on and what is off
- If the LLM, the online documentation and your help page differ, the help page of your own installation wins

 Tip: An LLM that looks something up has not always read the page (optional)

Some LLM tools can search the web or open a link you give them. That sounds like they read the page, but often they don't read all of it. Some tools pass the page through a second, smaller model that summarises it, and the main model only sees that summary. Claude Code works like this, for example (checked in October 2026; other tools may work differently). An answer can also be based only on the short snippets that a search engine shows for each result.

There is a good reason for this: a long web page is a lot of text to process, and a summary makes the answer faster and cheaper. So I don't expect this to disappear, even if the details change from tool to tool. And for many questions, a summary is all you need, for example "What is fastp used for?" or "Which tools are often used to trim reads?".

It becomes a problem when the detail is the point. A summary can drop a detail, mix up two options, or smooth over exactly the sentence that matters, and it reads just as convincingly as a correct answer. So when you need a default value, an option name or the version a page describes, open the page or the help page yourself and read the lines in question. Ask the LLM to quote the exact sentence it relies on, so you can find it and check it.

## Using an LLM to understand code

This is where I find LLMs most useful. You can paste a command and ask what each part does. This works well with free chat tools.

Some questions that work well:

Explain this bash command line by line. I am a biologist and new to the command line:

<paste the command here>

What happens to each file when I run this loop?

What happens if one of the input files is missing?

What is the difference between the `-o` and the `-h` option of `fastp`?

Keep in mind that the answer you get can still be wrong. An explanation is easier to check than a full script, though: you can test each claim with a small command or by looking at the help function yourself.

### Exercise

In the optional box at the end of the `fastp` section on day 2, we used this `awk` command (if you skipped that box, you don't have `stats_trimmed.txt`, but you can still do the checks in the answer on `stats_raw.txt`):

```
awk -F'\t' 'FNR > 1 {print $1, $4, $7}'  
↪ results/seqkit/stats_raw.txt results/seqkit/stats_trimmed.txt
```

1. Paste the command into UvA AI Chat (or another chat tool that you are allowed to use) and ask it to explain each part. This exercise is about practicing checking an answer from an LLM, not about getting the code answer.
2. Pick one thing the explanation claims, for example what `FNR > 1` does. How can you check this claim yourself, with the commands from day 1?

### Click me to see an answer

For question 2: the explanation should say that `FNR > 1` skips the first line of each file, the header. You can check this by comparing the input with the output:

```
# Look at the first lines of the input: the first line
↳ is the header
head -n 3 results/seqkit/stats_raw.txt

# Run awk on one file only: the header line should be
↳ missing
awk -F'\t' 'FNR > 1 {print $1, $4, $7}'
↳ results/seqkit/stats_raw.txt | head -n 3

# Count the lines: the awk output should have one line
↳ less than the input
wc -l results/seqkit/stats_raw.txt
awk -F'\t' 'FNR > 1 {print $1, $4, $7}'
↳ results/seqkit/stats_raw.txt | wc -l
```

You don't need to know awk to do this. You only need to know what the output *should* look like. This is also a good example for how knowing basic bash commands allows you to perform the suggested checks yourself, no LLM needed.

Another claim worth checking is what columns 1, 4 and 7 contain. When I asked ChatGPT (October 2026), its first answer showed an example row where column 7 was Q30. Further down, it did say that its examples were only illustrative. But if you skim a long answer, it is easy to miss that sentence and remember the example as a fact.

The header line of the file tells you what each column really is:

```
# Print only the header line
head -n 1 results/seqkit/stats_raw.txt
```

This prints:

```
file      format  type    num_seqs      sum_len min_len avg_len
max_len Q1      Q2      Q3      sum_gap N50      Q20(%) Q30(%)
AvgQual GC(%)
```

If you count the columns, column 1 is `file`, column 4 is `num_seqs` and column 7 is `avg_len`, the average read length. So the command prints the file name, the number of reads and the average read length, not a quality value.

What makes this mistake tricky is that the answer started out right. The command mentions `results/seqkit/`, and the example file in the answer began with the real seqkit column names: `file, format, type, num_seqs`. The example values for columns 2 to 6 also fit those columns. Only column 7 was suddenly a quality value. When the first part of an answer checks out, it is tempting to trust the rest as well. But an LLM doesn't get more reliable further into an answer, and one correct detail says nothing about the next one.

So check the detail that matters to you, not the first one you can check. And whenever an answer depends on your own files, look at the file yourself, for example with `head`.

💡 Tip: When the answer is longer than you want to read (optional)

LLM answers are often long. When I asked ChatGPT (October 2026) to explain the one-line awk command from the exercise above line by line, I got seven sections, two tables, example files and a summary at the end. I used ChatGPT without any memory of earlier chats on purpose, so it knew nothing about me except what I wrote in the question. Most of it was correct (the answer above shows one detail that was not). But the part I actually needed fits in one sentence, and it was near the top:

Read two files, skip the first line of each file, and print columns 1, 4, and 7 from every remaining line.

The rest explained things like what the folder `results/` is, and why a biologist might want to use this command. The few details that really matter, such as how FNR is different from NR, were somewhere in the middle. If you are new to the command line, it is hard to tell which parts are important and which are filler. And scrolling through a long, well-written answer can feel like learning, even when not much of it sticks. You don't need a perfect question to avoid this. You can simply say what is wrong with the answer. I replied that most students would not read through this much text, and the second answer was much shorter. Its core was this table:

| Part                             | Meaning                                                 |
|----------------------------------|---------------------------------------------------------|
| <code>awk</code>                 | Reads and processes text, one line at a time.           |
| <code>-F'\t'</code>              | Treats tabs as column separators.                       |
| <code>FNR &gt; 1</code>          | Skips the first line of each file (usually the header). |
| <code>print \$1, \$4, \$7</code> | Prints columns 1, 4, and 7.                             |
| <code>stats_raw.txt</code>       | First input file.                                       |

```
stats_trimmed.txt    Second input file.
```

That fits on one screen, and every line is something you can check. Other short follow-ups that work:

Shorter please, max 5 lines.

Only explain what `FNR > 1` does.

Which part of your answer should I check myself, and how?

Asking about one part at a time also makes each answer easier to check.

## Checking code with the commands from day 1

This is the main point of this page. Not all code needs the same level of understanding:

- **The analysis itself: understand every option.** The options of a tool like `fastp` decide what happens to your data: which reads are removed, and how much of each read is trimmed. These are scientific choices. You need to know what each option does and why you use it, whether you or an LLM wrote the command. Look them up with `fastp --help` or in the manual, and ask an LLM to explain an option if the manual is unclear
- **The code around it: know what it should do, then check.** Some code only organises the work: an `if/else` block that skips finished samples, or a `sed` or `awk` command that picks lines from a file. You may not understand every detail of how it works. But you must know what it is supposed to do to your files, and check that it really does

For these checks, the commands from day 1 are enough:

| Question                                   | Command                                                         |
|--------------------------------------------|-----------------------------------------------------------------|
| Did I get all the output files I expected? | <code>ls -l results/fastp, ls results/fastp \\<br/>wc -l</code> |
| Is any file empty?                         | <code>ls -l</code> (look for a size of 0)                       |
| Which samples will the loop work on?       | <code>cat samples.txt, wc -l samples.txt</code>                 |
| What does a wildcard match?                | <code>ls data/seq_project/*/*_R1.fastq.gz</code>                |
| How many reads are in a file?              | <code>zcat file.fastq.gz \\<br/>wc -l</code> (divided by 4)     |
| Does the output look right?                | <code>head, less -S</code>                                      |

| Question                                  | Command                                                   |
|-------------------------------------------|-----------------------------------------------------------|
| Did every sample show up in the log?      | <code>grep -c "Start fastp" logs/fastp_*.out</code>       |
| What will the loop do, before it does it? | put <code>echo</code> in front of the command (a dry run) |

Before you run anything, think about what you expect: how many samples, how many files, roughly how many reads. Then compare. If the numbers don't match, you have found a problem, even if you can't yet say where it is.

### Exercise

Imagine you ask an LLM how to count the reads in a fastq file, and it suggests:

```
zcat
↪ data/seq_project/barcode001_User1_ITS_1_L001/Sample-DUMMY1_R1.fastq.gz
↪ | grep -c "^@"
```

The idea: each read starts with a header line that begins with `@`, so counting those lines should count the reads (`^@` means “a line that starts with `@`”).

1. Run the command. How many reads does it report?
2. Count the reads in the same file the way we did on day 1. Do the two numbers match?
3. Can you think of a reason why they don't match? Hint: look at the fastq format again. Which other line of a read could start with `@`?

#### Click me to see an answer

On my computer (WSL on Windows), the `grep` command prints:

44882

The day 1 way:

```
zcat
↪ data/seq_project/barcode001_User1_ITS_1_L001/Sample-DUMMY1_R1.fastq.gz
↪ | wc -l
```

This prints:

179384

179,384 lines divided by 4 is 44,846 reads. So the `grep` command counts 36 reads too many.

The reason: the fourth line of each read holds the quality scores, and `@` is one of the characters used for a quality score. In 36 reads, the quality line happens to start with `@`, so `grep` counts these lines as well.

The command looked sensible, and the number looked plausible. Only the comparison with a second, simple count showed the problem.

## Worked example: an LLM writes a fastp job script

Let's look at a realistic example. I asked an LLM:

I work with samples S1-S100. I want to run fastp for quality trimming and remove reads below an average quality score of 20 and use the same cutoff for quality trimming of the reads themselves. Can you write a script to process all these samples at once on a HPC with slurm?

The prompt could be better, but it is roughly what many of us would type. This is the script it wrote (I only shortened the decorative lines in the comments):

```
#!/bin/bash
#SBATCH --job-name=fastp_trim
#SBATCH --array=1-100
#SBATCH --cpus-per-task=8
#SBATCH --mem=16G
#SBATCH --time=04:00:00
#SBATCH --output=logs/fastp_%A_%a.out
#SBATCH --error=logs/fastp_%A_%a.err

# -- Paths --
DATA_DIR="data"
OUT_DIR="trimmed"
mkdir -p "${OUT_DIR}" logs

# -- Sample name derived from array index --
SAMPLE="S${SLURM_ARRAY_TASK_ID}"

R1="${DATA_DIR}/${SAMPLE}_R1.fastq.gz"
```

```

R2="${DATA_DIR}/${SAMPLE}_R2.fastq.gz"

# -- Guard: skip if input files are missing --
if [[ ! -f "${R1}" || ! -f "${R2}" ]]; then
    echo "ERROR: Input files not found for ${SAMPLE}: ${R1} / ${R2}"
    ↪ >&2
    exit 1
fi

# -- Run fastp --
fastp \
    --in1 "${R1}" \
    --in2 "${R2}" \
    --out1 "${OUT_DIR}/${SAMPLE}_R1_trimmed.fastq.gz" \
    --out2 "${OUT_DIR}/${SAMPLE}_R2_trimmed.fastq.gz" \
    --average_qual 20 \ # discard reads with mean quality <
    ↪ 20
    --cut_mean_quality 20 \ # sliding-window trimming threshold
    --cut_front \ # trim low-quality bases from 5' end
    --cut_tail \ # trim low-quality bases from 3' end
    --thread 8 \
    --json "${OUT_DIR}/${SAMPLE}_fastp.json" \
    --html "${OUT_DIR}/${SAMPLE}_fastp.html"

echo "Done: ${SAMPLE}"

```

It looks professional: tidy comments, a check on the input files, separate log files per sample. It uses a **job array**, which we only saw in an optional box on day 2: SLURM starts the same script many times (here 100 times), and each copy gets its own number in the variable `SLURM_ARRAY_TASK_ID`. The script uses the long names of the fastp options (`--in1` instead of `-i`, `--out1` instead of `-o`); both forms do the same thing.

### **i** Exercise

Read the script carefully, as if you were going to run it on your own data. Write down everything you would check or change. Some questions to guide you:

1. Which fastp options will fastp actually receive? Look closely at the lines with a comment at the end
2. What does the script assume about your file names and folders? Would it work

with the files from this workshop?

3. Which fastp is used when the job runs on Crunchomics?
4. Where do the numbers for CPUs, memory and time come from?
5. Compare it with the fastp script from day 2. Is it wrong, or just different?

### Click me to see an answer

I sort the problems from most to least serious.

#### 1. It runs, but does the wrong thing

- **The comments at the end of the lines break the command.** The `\` only joins two lines when it is the very last character on the line (we saw this on day 2). Here, spaces and a comment follow it. The `\` then only protects the space after it, and the command ends at the end of that line. fastp receives the input and output files, `--average_qual 20`, and one extra argument that is just a space. Bash then tries to run the lines that follow as commands of their own, and each attempt fails with `command not found`. The `--thread` line has no comment, so its `\` works again, and `--thread`, `--json` and `--html` end up together in one failing command. So fastp never gets the trimming options, `--thread`, `--json` or `--html`. I missed this myself when I first read the script, which shows that reading is not enough: you also have to run it on one sample and look at the output. See the next exercise for a small test.
- **The job reports success anyway.** The last line, `echo "Done"`, always works. A job counts as successful when its last command works, so SLURM reports every task as finished without problems. The errors are only visible in the `.err` files. (With `set -euo pipefail` at the top of the script, it would have stopped at the first `command not found`, and SLURM would have reported the task as failed. fastp would still have run once with the incomplete options before that. See the tip on `set -euo pipefail` further down.)
- **The logs folder is made too late.** SLURM opens the log files before the script starts, so the `mkdir` for `logs` inside the script comes too late to matter. On Crunchomics, SLURM then makes a missing `logs` folder itself, but inside the folder where you typed `sbatch`, which is not always where you expect your logs (we saw this on day 2). Make the folder before you run `sbatch`, as we did on day 2, so that you know where your logs end up.

#### 2. It makes assumptions about your data

- **The file names are invented.** `S${SLURM_ARRAY_TASK_ID}` only works because the prompt said “S1-S100”. Real file names rarely look like that. In this workshop, the files are called `Sample-DUMMY1_R1.fastq.gz` and are in separate folders. A list file like `samples.txt` works with any names, and you can open it to see which samples will be processed.
- **It chose a trimming method for you.** The prompt was not clear about how to trim, so the LLM chose trimming from both ends of the read (without asking me for clarification). Is that what you want? `fastp 1.4.0` offers three methods, each with a window of 4 bases that moves along the read: `--cut_front` (from the start of the read, removes bases until the quality is good enough), `--cut_tail` (the same, from the end of the read) and `--cut_right` (from the start; at the first window with low quality, it removes that window and everything after it). Which one fits depends on your data. `--cut_mean_quality` only sets the quality threshold that these three methods use. On its own, it does nothing: `fastp` only trims by quality if you also add `--cut_front`, `--cut_tail` or `--cut_right`. (And 20 is already the default threshold, so the script would trim the same way without this line.) And `fastp` already filters reads by default, also without any of these options (see the exercise on default filtering above). Check `fastp --help` for the version you use, and ask yourself (or the LLM) why these options and not others.
- **The resources are guesses.** 8 CPUs, 16 GB of memory and 4 hours per sample are not based on your data. Asking for too much time means a longer wait in the queue. Run one sample first, then check what it actually used (the optional `sacct` box on day 2).
- **No software is activated.** There is no `conda activate`, so the script doesn’t say which `fastp` it needs. Which one the job uses then depends on your terminal at the moment you submit the job: SLURM normally copies its settings, including the `PATH`, into the job. If the `fastp_1.4.0` environment happened to be active, the job uses `fastp 1.4.0`. If not, it uses the `fastp 1.0.1` of Crunchomics. Your results should not depend on such an accident.

### 3. It is harder to read than needed

- `DATA_DIR` and `OUT_DIR` hide two short folder names. You have to look them up to know where the files go. Variables like `SAMPLE`, `R1` and `R2`, however, are useful, because they are used many times.

- The one place where a variable *would* help is the number 8: it appears twice (`--cpus-per-task=8` and `--thread 8`). If you change one, you have to remember the other. `--thread ${SLURM_CPUS_PER_TASK}` would always use the number of CPUs you requested.
- The check on the input files looks safe, but it is easy to overlook when it fires. If a file is missing, only that one task stops. Its error message ends up in one `.err` file among many, and the other tasks finish normally. With 4 samples you might notice. With 1000 samples, one failed task is easy to miss, and you continue with 999 samples without knowing it. I personally prefer checks that fail loudly, before anything runs: check that all input files exist *before* you submit the job, with `wc -l samples.txt` and a loop that runs `ls` on every R1 and R2 file. If a file is missing, you see it right away and the job never starts. And after the job, count the output files, whatever checks the script contains.

#### 4. Different from day 2: wrong, or just different?

The LLM used a job array, while we used a for-loop inside one job script. Both are fine. An array runs the samples at the same time, a loop runs them one after the other. Something different from what you learned is not wrong by itself. But you should understand it before you use it. Also be careful with arrays on a shared HPC: `--array=1-100` with 8 CPUs each can ask for 800 CPUs at once. `--array=1-100%8` runs at most 8 tasks at the same time.

You can test the first problem without `fastp` and without an HPC. Put `echo` in front of the command, as in a dry run, and add comments at the end of the lines like the LLM did. Save these lines in a script called `comment_test.sh`:

```
echo fastp \
  --in1 Sample-DUMMY1_R1.fastq.gz \   # the R1 file
  --in2 Sample-DUMMY1_R2.fastq.gz \   # the R2 file
  --thread 2
echo "Done"
```

Then run it with `bash comment_test.sh`. On my computer (WSL with Ubuntu), this prints:

```
fastp --in1 Sample-DUMMY1_R1.fastq.gz
comment_test.sh: line 3: --in2: command not found
comment_test.sh: line 4: --thread: command not found
```

Done

The first line shows the full command that fastp would receive: only the R1 file. The R2 file and `--thread` never reach it, and bash simply continues with the next lines. In the worked example, this is what makes the mistake dangerous: fastp still gets the input and output files and `--average_qual 20`, so it runs, writes trimmed files, and SLURM reports success. The output looks normal, but it was made without the trimming options you asked for.

## Less code is often better

LLMs have learned from a lot of code written for large, automated software systems. There, a script should keep running no matter what goes wrong. In a scientific analysis, you usually want the opposite: if something goes wrong, you want to know.

A common example is a check that skips a sample when its input file is missing. Here, I made a typo in `samples.txt` (Sample-DUMY3 instead of Sample-DUMMY3) and ran a loop in the style that LLMs often write. To keep it short, the loop only prints which sample it would work on:

```
for sample in $(cat samples.txt); do
    R1=$(ls data/seq_project/*/${sample}_R1.fastq.gz 2>/dev/null)
    if [ ! -f "$R1" ]; then
        continue
    fi
    echo "Processing $sample"
done
echo "All samples done"
```

On my computer, this prints:

```
Processing Sample-DUMMY1
Processing Sample-DUMMY2
Processing Sample-DUMMY4
All samples done
```

`continue` tells the loop to skip the rest of this round and go on with the next sample. The loop skipped sample 3 without a single word, and then reported that all samples were done. If this were a long fastp job, the results would look complete, but cover only 3 of the 4 samples. `2>/dev/null` makes it worse: it throws away the error message of `ls`.

Compare this with the simple loop from day 2 (here with `ls` instead of fastp):

```

for sample in $(cat samples.txt); do
    echo "Start: $sample"
    ls data/seq_project/*/${sample}_R1.fastq.gz
done

```

This prints:

```

Start: Sample-DUMMY1
data/seq_project/barcode001_User1_ITS_1_L001/Sample-DUMMY1_R1.fastq.gz
Start: Sample-DUMMY2
data/seq_project/barcode002_User1_ITS_9_L001/Sample-DUMMY2_R1.fastq.gz
Start: Sample-DUMMY3
ls: cannot access 'data/seq_project/*/Sample-DUMMY3_R1.fastq.gz': No such
file or directory
Start: Sample-DUMMY4
data/seq_project/barcode004_User2_ITS_9_L001/Sample-DUMMY4_R1.fastq.gz

```

The simple loop has no check at all, and that is exactly why the problem is visible: the error message appears right below the sample name, and ends up in the log file of a job. Together with a check of the output (4 samples in, 16 files out), you would notice the missing sample right away.

So knowing your own data and checking it with basic commands often works better than adding checks to the code. Before you submit a job, check the inputs with `cat samples.txt`, `wc -l samples.txt` and an `ls` loop like the one above. After the job, count the outputs.

#### Tip: Skipping samples that are already done (optional)

Sometimes a check in the loop does make sense. Say a job stopped halfway, and you don't want to run fastp again for the samples that are already done. Then write the check so that you can read it, and so that every skip shows up in the log:

```

for sample in $(cat samples.txt); do
    if [ -s results/fastp/${sample}_R1.trimmed.fastq.gz ]; then
        echo "Skip $sample: output already exists"
    else
        echo "Run fastp for $sample"
    fi
done

```

- `if [ ... ]; then ... else ... fi`: if the test between the square brackets

is true, bash runs the commands after **then**. Otherwise, it runs the commands after **else**. **fi** (if spelled backwards) closes the block

- **-s**: true if the file exists *and* is not empty. A job that crashed can leave an empty file behind, which **-f** (true if the file exists) would accept
- Neither **-s** nor **-f** checks that a file is complete. A job that ran out of time can leave a half-written file. When in doubt, delete the outputs of the sample that was running when the job stopped, and run it again

Replace the **echo** after **else** with the **fastp** command once the dry run looks right. The IBED bioinformatics pages explain this in more detail, each with a note on code written by LLMs:

- **if/else, continue and break**: checking input files
- **set -euo pipefail**: a line that LLMs often put at the top of a script. It makes a script stop at the first command that fails. That sounds like exactly the loud failure I recommend on this page, and in the worked example it would have helped. But it also stops a script for things that are not errors. For example, **grep** counts as failed when it finds no match, even if “no match” is a perfectly fine answer. While you develop a script, this makes it harder to see what is going on. So leave it out while you develop, and only consider adding it once the script works and you know which of your commands can “fail” without a real problem

### **i** Exercise

Take the LLM script from the worked example, and change it into a script that works for the data of this workshop, in the style of day 2. Keep the job array if you like, or use a for-loop.

Then answer: which lines did you remove, and why? Which lines did you add?

#### **Click me to see an answer**

There are several good answers. This one keeps the array, and takes the sample name from **samples.txt**. Make the **logs** folder before you submit it.

```
#!/bin/bash
#SBATCH --job-name=fastp_array
#SBATCH --output=logs/fastp_array_%A_%a.out
#SBATCH --error=logs/fastp_array_%A_%a.err
#SBATCH --array=1-4
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#SBATCH --mem=5G
#SBATCH --time=01:00:00

# Make conda available in the job, and activate the
↪ environment with fastp
source ~/.bashrc
conda activate fastp_1.4.0

# Take the line with the number of this task from
↪ samples.txt
sample=$(sed -n "${SLURM_ARRAY_TASK_ID}p" samples.txt)
echo "Start fastp for: $sample"

fastp \
  -i data/seq_project/*/${sample}_R1.fastq.gz \
  -I data/seq_project/*/${sample}_R2.fastq.gz \
  -o
  ↪ results/fastp_array/${sample}_R1.trimmed.fastq.gz
  ↪ \
  -O
  ↪ results/fastp_array/${sample}_R2.trimmed.fastq.gz
  ↪ \
  -h results/fastp_array/${sample}.html \
  -j results/fastp_array/${sample}.json \
  --average_qual 20 \
  --thread ${SLURM_CPUS_PER_TASK}
```

- `--array=1-4`: one task per line in `samples.txt`. Check the number with `wc -l samples.txt`
- `sed -n "${SLURM_ARRAY_TASK_ID}p" samples.txt`: prints only line number `SLURM_ARRAY_TASK_ID` of the file. Task 1 gets the first sample, task 2 the second, and so on

- `--thread ${SLURM_CPUS_PER_TASK}`: SLURM stores the number of CPUs you requested with `--cpus-per-task` in this variable. `fastp` now always uses as many CPUs as the job has, also if you change the number later
- Removed: `DATA_DIR`, `OUT_DIR`, the check on the input files (check the inputs before you submit instead), the comments at the end of the lines and `echo "Done"`
- Left out on purpose: the quality trimming. The prompt asked for it, but not for a specific method. `fastp` has three ways to trim reads by quality (`--cut_front`, `--cut_tail` and `--cut_right`, see the answer to the worked example and `fastp --help`). Read about them first, decide which one fits your data, and only then add it, with a note in your notebook on why you chose it
- Added: `conda activate`, `samples.txt`, and our own names for the reports
- Make the output folder before you submit: `mkdir -p logs results/fastp_array`

Before you run it on all samples: set `--array=1` to run only the first sample, and check the output and the `fastp` report.

## When your code does not work

When your code crashes or gives strange results, it is tempting to paste everything into an LLM and ask it to fix it. I suggest a different order:

1. **Read the error message.** As on day 1: which command complains, and about which file? Often, the message already tells you what is wrong
2. **Narrow it down.** Bugs are almost always small and in one place. Run the steps one by one. Put `echo` in front of a command to see what bash actually runs, or `echo $sample` to see what is in a variable
3. **Then ask a specific question.** Give the LLM the one command that fails, the exact error message and a few lines of the input (replace anything sensitive, see above). Ask what the error means, not for a new script

A rewritten script may run without errors, but you then have to check all of it again, and you have learned nothing about why the first version failed.

## Further reading

- [On GenAI usage](#), at the end of the IBED for-loops page: what to look out for in loops written by an LLM
- The UvA's [AI literacy course for students](#): what generative AI is, how it works and what its limits are (about 45 to 60 minutes)
- [UvA AI Chat](#) and the UvA [rules for students](#)