# Toolbox: text & data — jq, sd, pandoc

Key takeaways

1. Pick the tool by data shape: jq for JSON, sd for text, pandoc for docs.
2. Single-quote every jq program and add -r when the shell consumes it.
3. sd rewrites files in place with no -i, so preview with -p first.

# One question: what shape is the data?

jq owns fields, sd owns text, pandoc owns documents

## jq

Structured JSON: query, reshape, build. When the data has fields, not lines.

**Pipe JSON into jq, never into grep.**

## sd

Find and replace, literal or regex, across one file or a whole tree. When you're editing text.

**Preview with -p before sd rewrites in place.**

## pandoc

Convert document formats: md, pdf, docx, html. When the unit of work is a document.

**Convert in one pandoc call, not by hand.**

Wrong-tool tell: grep and sed hunting through JSON, or hand-editing what pandoc converts in one line.

# Toolbox — structured JSON

## jq

A JSON processor: filter, reshape, and build JSON right from the shell.

Replaces: grep / sed on JSON

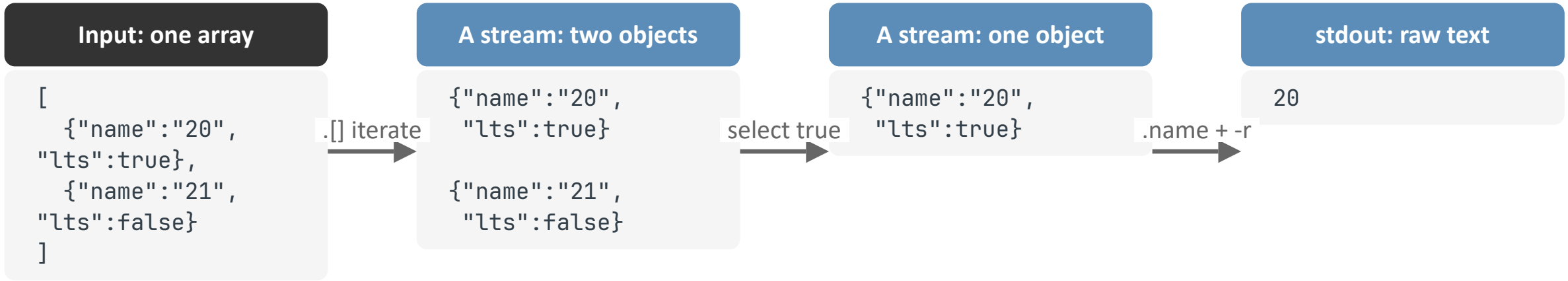
```
# -r: one field as raw text
jq -r '.tag_name' rel.json

# List names, one per line
jq -r '.[]|.name' n.json

# -n: build JSON from nothing
jq -n '{ok:true}'
```

# jq filters pass values, not lines

Follow one program from a JSON array to raw shell text




jq is a value pipeline: each filter consumes the stream on its left and emits the stream on its right.

# jq: four moves an agent actually makes

An agent hits JSON constantly — API responses, config, tool output. These four cover almost all of it.

### Filter + -r: raw text for the shell

```
jq -r '.[]
  | select(.lts)
  | .name' rel.json
```

### -n: build JSON from nothing

```
jq -n \
  --arg id "$SID" \
  --arg name "$N" \
  '{name:$name, id:$id}'
```

### @tsv: columns for cut / awk

```
jq -r '[.name, .id,
  .last_used]
  | @tsv' *.json
```

### -e: assert in a pipeline (exit code)

```
jq -e '.type=="result"
  and (.is_error|not)' \
  out.json
```

# jq gotcha: single-quote the program

Double quotes let the shell eat $, backticks, and * before jq ever sees the program. Single-quote it, and use -r when the output feeds another command.

**double quotes — shell mangles it**

```
# $key expanded by the
# shell, not jq
jq ".files[] | .$key"

# no -r: "1.2" with quotes
jq .version
```

**single quotes + -r**

```
# program reaches jq intact
jq '.files[] | .name'

# raw 1.2, empty if absent,
# safe to pipe onward
jq -r '.version // empty'
```

# Toolbox — text find & replace

## sd

Find & replace with real regex: capture groups and a syntax you don’t have to fight.

Replaces: sed s/old/new/

```
# Rewrite the file in place
sd TODO DONE notes.txt

# No file: replace on stdin
echo 07-06 | sd '-' '/'

# Tabs to spaces in a pipe
git diff | sd '\t' '  '
```

# sd: replace that reads like you think

No slashes to escape, no s/pat/rep/g ceremony. Regex by default; $1 $2 for capture groups in the replacement.

### Reorder with capture groups

```
echo 'Witchel, Emmett' |
  sd '(\w+),\s+(\w+)' \
     '${2} ${1}'
```

### -F: literal string, no escaping

```
sd -F 'utils.old' \
      'utils.new' \
      src/app.py
```

### Pipe: no file = read STDIN

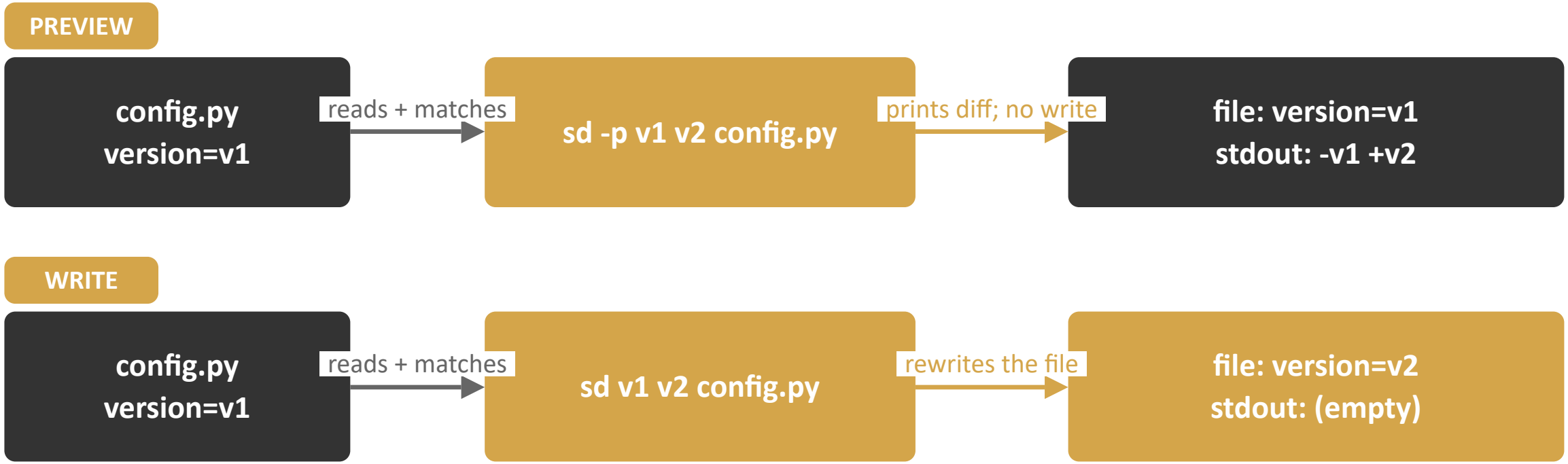
```
cat build.log |
  sd '\d+ms' 'N ms'
```

### -n 1: only the first match

```
sd -n 1 'v0' 'v1' \
   CHANGELOG.md
```

# -p previews; without it, sd writes

The match and replacement stay the same; the side effect changes




Same match and replacement. -p changes the side effect: a diff instead of a write.

# sd gotcha: it writes in place — preview first

Give sd a file and it rewrites that file, no -i needed (unlike sed). Preview with -p before any bulk run; it’s regex by default, so -F when you mean a literal.

**fires across every file, instantly**

```
# every file rewritten;
# $1_beta is also ambiguous
# (digit runs into _beta)
sd 'v(\d)' 'v$1_beta' \
   src/**/*.py
```

**-p to preview, brace the group**

```
# -p shows the diff, writes
# nothing; ${1} ends the
# group; one file, on purpose
sd -p 'v(\d)' 'v${1}_beta' \
   src/config.py
```

# Toolbox — document conversion

## pandoc

Convert documents between formats — markdown, PDF, Word, HTML, and dozens more.

Replaces: copy-paste reformatting by hand

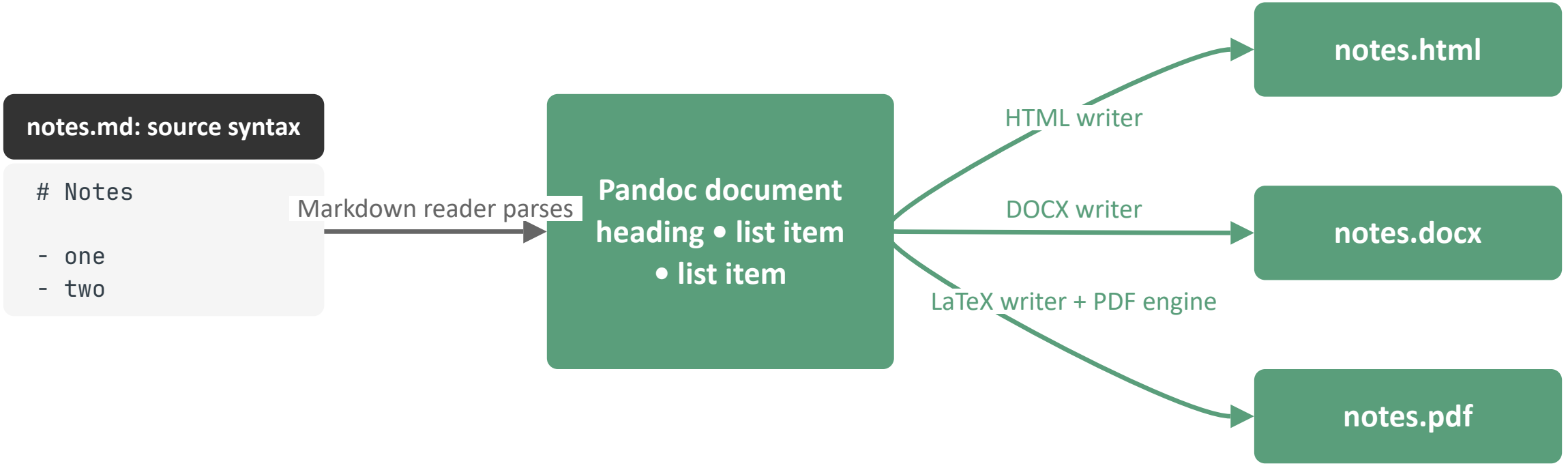
```
# md to PDF (-o picks the format)
pandoc notes.md -o notes.pdf

# md to Word
pandoc draft.md -o draft.docx

# HTML to md, on stdout
pandoc page.html -t markdown
```

# pandoc parses once, then chooses a writer

The output extension selects the HTML, DOCX, or PDF path




The reader preserves document structure; the chosen writer decides how that structure appears in the target format.

# pandoc: three conversions you'll reach for

-o names the output format; one command replaces a reformatting afternoon.

**share a clean PDF (needs a LaTeX engine)**

```
pandoc notes.md -o notes.pdf
```

**hand off to a Word / track-changes collaborator**

```
pandoc draft.md -o draft.docx
```

**scrape a web page into your markdown notes**

```
pandoc -f html -t markdown page.html > notes.md
```

**PDF output shells out to LaTeX. No TeX → install a TeX engine, or emit .html / .docx instead.**

# Key Takeaways

1. Pick by shape: fielded JSON → jq, text find/replace → sd, whole documents → pandoc.
2. jq: single-quote the program, -r to feed the shell, -n to build JSON from scratch.
3. sd rewrites files in place with no -i — -p to preview, -F for literal; pandoc md→pdf needs a LaTeX engine.