

The Well-Set Typographer

**Typography, Typesetting, and Document Production from the
Command Line**

Vijay Mathew

April 13, 2026

Table of contents

	1
Introduction: Why This Matters, and Why Here	3
What typography actually does	4
Why the command line?	4
Who this book is for	7
How to read this book	8
Letters, Ink, and Pixels: A Foundation	9
A Brief History of Typesetting	11
Phototypesetting and the desktop revolution	12
Knuth, TeX, and why it matters	14
The web and the fracturing of print	16
Typography Fundamentals	21
Anatomy of a typeface	21
Classifying typefaces	25
Spacing: kerning, tracking, and leading	28
The typographic scale	31
Hierarchy, contrast, and the reader's eye	32
Digital Typesetting Concepts	35
Units of measurement	35
Font formats	37
Rasterization and hinting	39
Print versus screen	42
Unicode, encoding, and OpenType features	44

Font Management from the CLI	47
How Linux finds fonts: fontconfig	47
fc-list: surveying what you have	49
fc-match: resolving font requests	51
fc-query and fc-scan: inspecting font files	52
Installing fonts manually	54
Previewing fonts without a GUI	56
Font licensing basics	58
Markdown as Source: The Modern Workflow	63
Markdown for Document Authors	65
The Markdown landscape: CommonMark, GFM, and Pan- doc Markdown	65
Front matter and metadata with YAML	67
Basic Markdown syntax	70
Footnotes	72
Citations	73
Tables	76
Cross-references	78
Definition lists and other block elements	79
Structuring long documents	81
Pandoc — The Universal Converter	85
How Pandoc thinks: the AST model	85
The basic invocation	87
Essential flags	88
PDF engines and when to use each	91
Templates	92
Defaults files	95
Lua filters	96
Practical conversion patterns	99
Generating HTML	101
What Pandoc’s HTML output looks like	101
Semantic HTML and accessibility	103

CSS for web typography	105
Syntax highlighting	112
Tables of contents	113
Single-file vs multi-page documents	115
Generating Print-Ready PDFs	119
The three PDF pipelines	119
Page geometry in print-oriented PDF workflows	120
Headers and footers	123
Font embedding and verification	124
PDF metadata and links	126
Colour management for print	128
Bleed and crop marks	129
The wkhtmltopdf pipeline	130
Debugging backend-specific PDF failures	132
A complete print-ready PDF workflow	134
EPUB and Ebooks	137
What EPUB is	137
Generating EPUB with Pandoc	139
EPUB metadata	141
CSS for EPUB	142
Validation with epubcheck	146
Reflowable versus fixed-layout EPUB	147
Kindle and distribution formats	148
A practical EPUB build command	150
Build Systems and Automation	153
Shell scripts for simple builds	153
Make: the right tool for document builds	156
Version control for prose	161
Automating builds with CI/CD	165
Putting it together: a complete project scaffold	169

The Toolbox	171
Choosing the Right Tool	173
Three axes of decision	173
The tools and their niches	175
The comparison matrix	178
A decision flowchart	179
Hybrid workflows	180
A note on staying current	181
LaTeX	183
TeX distributions	183
Document structure	185
Engines: pdfLaTeX, XeLaTeX, LuaLaTeX	186
Mathematics	188
Essential packages	190
Bibliography management	193
Custom commands and environments	194
Custom document classes	196
The compilation workflow with latexmk	198
Typst — The Modern Alternative	201
What Typst is	201
Installation	202
Basic syntax	203
Set and show rules	204
Typography configuration	206
Mathematics	207
Scripting and programmatic documents	209
Templates and reuse	212
Bibliography and citations	213
The package ecosystem	214
Current limitations	215
Compilation workflow	216
Quarto	219
Installation and setup	220

The QMD format	221
Executable code blocks	222
Inline code and computed values	223
Project types	224
Callouts and special blocks	228
Themes and customisation	229
Extensions	231
Code execution and reproducibility	231
Comparison with R Markdown	232
A complete project workflow	233
Emacs and Org Mode	237
Should you learn Emacs for Org Mode?	237
The Org format	238
The export system	241
Babel: literate programming in Org	246
AUCTeX: LaTeX editing in Emacs	248
A practical Org workflow	250
groff, troff, and the Unix Heritage	253
The *roff family	253
The groff command	255
The -ms macros for documents	256
Writing man pages	258
Pandoc as a man page authoring tool	263
groff for documents today	265
Diagrams and Figures from the CLI	267
Choosing the right format	267
Graphviz: graphs and networks	268
TikZ: programmatic drawing in LaTeX	271
Mermaid: diagrams from text in web contexts	273
D2: a modern alternative to DOT	275
Data plots with gnuplot and Python	276
Integrating diagrams in documents	278

Document Gallery: Real Examples	283
Letters and Correspondence	285
What a letter requires	285
Typst as the default formal-letter template	286
Postal-compliance edge cases	288
Cover letters with a personal header	288
A Markdown source for flexible formats	289
Mail merge: one letter per recipient	290
The Typst letter template	293
Choosing the approach	295
Résumés and CVs	297
What a CV requires typographically	297
A single-column résumé: Markdown source, Typst PDF	298
A two-column layout	300
The academic CV	302
The primary workflow: single source, multiple outputs	303
The Typst CV	307
Choosing the approach	308
Articles and Reports	311
A journal-style article	311
The primary Pandoc or Quarto article workflow	317
A technical report	319
A two-column magazine layout	323
Building articles from Markdown with Typst or LaTeX	325
Presentations	327
Reveal.js as the default deck	327
Markdown source shared across slide targets	331
Reveal.js: HTML presentations	333
Producing from a common source	335
Quarto for slides with executable code	335
Books — The Complete Project	339
Project structure	339

The PDF backend: Typst first, LaTeX when needed	341
Running headers	343
Index generation	344
Bibliography	345
The Pandoc multi-format build	346
Heading levels and shift-heading-level-by	349
The colophon	349
Craft and Refinement	351
Tables and Complex Layouts	353
Markdown and backend-aware table workflows	353
Table alignment in CSS	356
Sidebars and callout boxes	357
Templates and Style Systems	359
Building a reusable PDF style layer	359
Pandoc templates	361
CSS custom property systems for web typography	363
Multilingual and Non-Latin Typesetting	369
Multilingual Latin-script documents	369
Bidirectional text: Arabic and Hebrew	371
CJK typesetting: Chinese, Japanese, Korean	372
OpenType language features	373
Fine Typography	375
Backend-specific microtypography	375
Optical margin alignment	378
Widows and orphans	378
Hyphenation control	379
The paragraph as the unit of design	380
Letterspacing and small capitals	380
Typographic details in Markdown and Pandoc output	381

Appendices	383
Appendix A: Quick Reference — Pandoc Flags	383
Input and output	383
Document structure	384
Metadata and variables	384
Templates and styling	385
PDF generation	385
Citations and bibliography	386
Filters and extensions	386
Code highlighting	387
EPUB options	387
Diagnostics and utilities	388
Commonly used combinations	388
Appendix B: Essential LaTeX Packages	391
Page layout and geometry	391
Typography and fonts	392
Mathematics	393
Tables	393
Figures and floats	394
Lists	394
Cross-references and hyperlinks	395
Bibliography	395
Code and verbatim	396
Colour and boxes	396
Language support	397
Indexing	397
Diagrams and graphics	397
Document classes (not packages)	398
Recommended minimal preambles	398
Appendix C: Tool Comparison Matrix	401
Primary tools: comprehensive comparison	401
PDF output engines	402
Output format support by tool	402
Source format compatibility	403

Decision guide: which tool for which job	403
Diagram tools comparison	404
Font management tools	405
Build tool comparison	406
Appendix D: Free Font Resources	407
Serif typefaces for body text	407
Sans-serif typefaces	410
Monospace typefaces	411
Mathematical fonts	413
CJK fonts	414
Finding and installing fonts	415
Appendix E: Further Reading and Bibliography	417
Typography and type design	417
TeX and LaTeX	418
Pandoc and Markdown	419
Document engineering and build systems	419
Web typography and CSS	420
History of printing and typography	421
Online resources	421
Bibliography	422

Introduction: Why This Matters, and Why Here

Look at the last document you produced — a report, an email attachment, a presentation you converted to PDF before sending. Look at it carefully. Not at the content: at the form. Are the lines too long? Do the headings follow a consistent hierarchy, or do they vary in size because you adjusted them by eye? Are the margins the same on every page? Does the typeface suit the content, or was it the default, which is to say no decision was made at all? Is the spacing between paragraphs consistent, or does it vary because you pressed Return twice in some places and once in others?

If the document was produced in a word processor, the answer to most of these questions is probably uncomfortable. This is not an accident, and it is not your fault. It is the predictable result of a production tool that conflates *writing* with *typesetting* — that makes every typographic decision available at every moment, that rewards adjustment over principle, and that treats the visual appearance of a document as a series of individual choices rather than as the expression of a coherent underlying system.

Good typography does not work that way. It works through systems: through decisions made once and applied consistently, through rules that govern the relationships between elements rather than the properties of each element in isolation, through the kind of structural thinking that produces documents that look right not because someone made them look right, but because the system that generated them is itself right. This book is about building those systems from the command line.

What typography actually does

Typography is not decoration. It is not the choice between Arial and Helvetica, or the question of whether a heading should be 18-point or 20-point. It is the craft of making language visible — of translating the structure, tone, and hierarchy of written content into a visual form that a reader can navigate without effort.

When typography works well, it is invisible. The reader moves through the document absorbing the content, and the form is not a presence but an absence of obstacles. The lines are the right length — long enough to establish rhythm, short enough that the eye does not tire finding the start of the next line. The typeface has the right voice — neutral enough not to intrude, characterful enough not to be blank. The spacing gives the text room to breathe without leaving it stranded in emptiness. The hierarchy is clear — the reader always knows where they are in the document and how the current section relates to what came before.

When typography fails, the reader feels it, even if they cannot name it. The text is too dense and they slow down. The headings are too large and they feel like they are being shouted at. The lines are too long and their eyes get lost crossing to the next one. The spacing is irregular and the page looks unfinished. None of this prevents reading. But it adds friction, and friction accumulates, and documents that accumulate enough friction do not get read to the end.

This is not an aesthetic matter, though aesthetics are involved. It is a functional matter. Typography is the quality of the interface between your words and your reader's attention. Better typography means your words get through. Worse typography means some of them don't.

Why the command line?

The central claim of this book is that the command line is the right place to produce serious documents. Not the only place — context matters, and the book is honest about when GUI tools are appropriate — but the natural home for documents where quality and reproducibility matter.

This claim requires justification, and the justification is not primarily about preferring terminals to graphical interfaces. It is about what the two kinds of tool fundamentally do.

A word processor treats text and appearance as inseparable. When you write a heading in Microsoft Word, you are simultaneously entering content and making a typographic decision. The heading's size, weight, and spacing are properties you can see and adjust. This feels like control. What it actually produces is an ad hoc design process where typographic decisions are made by whim, habit, and imitation — by pressing Heading 1 because that is what Heading 1 looks like, not because you have thought about what a heading should be in this document.

A CLI workflow separates writing from typesetting. You write in plain text. The text contains structure — headings marked with #, paragraphs separated by blank lines, emphasis marked with * — but no appearance. The appearance is defined elsewhere, in a template or stylesheet or document class, and applied by a compilation step. The writing and the typesetting are decoupled. This decoupling is what makes quality achievable.

It is the same principle that separates content from presentation in HTML and CSS, or that separates data from display in any well-designed software system. When you write a stylesheet that says all first-level headings are 1.75rem, set in Fira Sans, with 1.5em of space above and 0.5em below, you are making a typographic decision once and applying it to every heading in every document that uses that stylesheet. Change it once; it changes everywhere. This is a system.

When you define a Typst template, a Quarto project format, or a LaTeX document class that sets the margin, the typeface, the baseline grid, and the section heading format, every document that uses that layer inherits those decisions. The author writes; the system typesets. The author's attention is on the argument, not on whether this heading is a little too close to the text below it.

The CLI enforces reproducibility. A document produced from a Markdown source file, a template, and a make command will be identical every time you build it, on any machine, by any person, now and in five years.

The same inputs produce the same output. This is not true of a Word document, which may look different depending on the version of Word, the operating system, the installed fonts, the screen resolution, and a dozen other environmental factors that are invisible until they break something.

Reproducibility matters for more than correctness. It matters for collaboration. It matters for automation. It matters for maintenance — for the ability to revise a document years later and know that the revisions will be applied consistently. It matters for the kind of long-term work that word processors actively resist: the book that takes three years to write, the documentation set that covers a hundred software components, the academic paper that passes through a dozen revisions and four co-authors before submission.

The CLI integrates with everything. A document source file in Markdown or LaTeX is a text file. It can be compared with `diff`. It can be version-controlled with `git`, with meaningful line-by-line change tracking. It can be generated programmatically, assembled from components, tested for compliance with house style, built in parallel with other documents, and deployed automatically when the source changes. None of this is possible with a binary file that encodes its content and appearance together in a proprietary format.

The CLI produces better output. This is a bolder claim, but it is also a true one, and it is worth being direct about it. A print-focused CLI workflow using Typst or TeX-class engines produces better-justified text than any word processor. Not marginally better — substantially better, in a way that is visible to any trained eye and perceptible as a vague sense of ease and quality even to untrained readers. The reason is architectural: these systems reason about the paragraph and the page as composition problems, not as ad hoc visual edits. TeX's paragraph-level line-breaking algorithm, unchanged in essential respects since 1978, remains the historical benchmark. Word processors break greedily, one line at a time, producing good individual lines but suboptimal paragraphs.

TeX's microtype package introduced a level of freely available microtypographic control that word processors still do not match. Modern PDF-first tools such as Typst bring much of the same seriousness to line breaking and page composition without asking the author to live inside TeX macros. System-font access to OpenType features — old-style figures that sit in running text the way lowercase letters do, true small capitals that match the weight of the surrounding text, discretionary ligatures that the font designer chose to include — becomes a configuration decision, not a manual formatting exercise.

These are not minor niceties. They are the difference between a document that is typeset and a document that happens to contain text.

Who this book is for

This book is for writers and researchers who produce documents that matter — academic papers, technical reports, books, documentation, correspondence — and who want to produce them well. It assumes you are comfortable with a terminal and a text editor. It does not assume you know LaTeX, or Pandoc, or any of the other tools it covers. It teaches them.

The book is also for people who already use some of these tools but want a more complete picture: the LaTeX user who doesn't know Pandoc, the Pandoc user who doesn't know what LaTeX is doing behind the scenes, the developer who sets up documentation pipelines without knowing what makes the typographic output good or bad.

It is not a book about typography as an end in itself. It is a book about producing documents that work: that communicate clearly, look professional, and can be maintained, revised, and built automatically. Good typography is in service of those goals, not separate from them.

How to read this book

Part I covers the conceptual foundations: the history of typesetting, the principles of typography, and the technical vocabulary of digital type. Readers who are impatient to get to the tools can skim this part and refer back to it when specific terms come up, but the concepts in it will make everything that follows more comprehensible and more usable.

Part II covers the Markdown-to-everywhere workflow: Pandoc, Markdown conventions, and the generation of HTML, PDF, and EPUB from a single source. This is the practical core of the book for most readers.

Part III is a survey of the toolbox: LaTeX as the historical standard and compatibility layer, Typst as the preferred PDF-first tool for new work, Quarto as the computational and multi-format system, Emacs and Org Mode, groff, and the diagram tools that generate figures and illustrations. Each tool is covered with enough depth to use it for real work.

Part IV is a gallery: five document types worked through from start to finish. Letters, résumés, articles, presentations, and books. Each chapter in this part is a complete example that can be used as a template.

Part V covers the craft layer: tables and complex layouts, templates and style systems, multilingual typesetting, and fine typography. These are the topics that separate adequate documents from excellent ones.

The appendices contain reference material: Pandoc flags, LaTeX packages, a tool comparison matrix, free font resources, and a bibliography.

Documents carry authority or fail to. They invite trust or undermine it. They respect the reader's time or waste it. Typography is the mechanism by which they do these things, and the command line is where that mechanism is most precisely controllable. That is why this matters, and that is why here.

Letters, Ink, and Pixels: A Foundation

A Brief History of Typesetting

Before there were fonts, there were hands. For most of human history, if you wanted a copy of a document, you found someone with good handwriting and sufficient patience and you waited. The result was beautiful, often — medieval illuminated manuscripts rank among the finest visual artifacts ever produced — but it was also slow, expensive, and error-prone. Every copy introduced new mistakes. Every scriptorium was a bottleneck. Knowledge moved at the speed of a quill.

What Johannes Gutenberg did in Mainz around 1450 was not, strictly speaking, invent printing. Woodblock printing had existed in China for centuries. What he invented was a *system*: individual metal letters, cast from an alloy of lead, tin, and antimony, that could be arranged into words, locked into frames, inked, and pressed against paper — then broken apart and rearranged for the next page. The key insight was reusability. Each letter was a small machine, manufactured once and used thousands of times.

This system gave us most of the vocabulary we still use today. The *type case* was a wooden tray divided into compartments, one per character. Capital letters were stored in the upper case; small letters in the lower — which is why we call them uppercase and lowercase to this day. *Leading* was the thin strip of lead placed between lines to create vertical spacing. The *em* was the width of the letter M, used as a unit of horizontal measure because M was the widest character in most typefaces; the *en* was half that. *Points* and *picas* were standardized units of measurement developed over the following centuries to let printers specify type sizes and column widths with precision.

These terms did not fade away when the technology changed. They migrated into every subsequent system — phototypesetting, desktop publishing, CSS, LaTeX, Pandoc — and you will encounter them

throughout this book. When you write `font-size: 12pt` in CSS or `\setlength{\parindent}{1em}` in LaTeX, you are using units invented to describe the dimensions of metal objects that haven't been manufactured for fifty years. The history is embedded in the syntax.

For four centuries, hot metal type was how the world's text was made. The craft evolved — typefaces multiplied, presses grew more powerful, paper became cheaper — but the fundamental process stayed the same: a compositor stood at a type case and assembled text by hand, one letter at a time. A skilled compositor could set perhaps 1,500 characters per hour. A single page of a novel might contain 3,000. It was exacting, physical work.

The Linotype machine, patented by Ottmar Mergenthaler in 1886, changed the speed of composition without changing its material nature. The operator sat at a keyboard — the first time in the history of printing that you typed to produce type — and the machine cast entire lines of text in a single slug of hot metal. Hence the name: line-o'-type. A Linotype operator could set five or six times faster than a hand compositor. Newspapers, which needed to produce thousands of words of fresh text every night, adopted it immediately. The Linotype hum became the sound of the press room.

What is worth holding onto from this era — worth holding onto for the whole book — is the attitude it engendered toward the text on the page. A printer who had physically assembled each line of a paragraph, who had chosen each piece of metal and placed it by hand, who had locked the forme and pulled the proof and corrected it and reprinted it, had a relationship with the text that was intimate and exacting. Every decision about spacing, every choice of typeface, every adjustment of leading was a deliberate act performed by a person who understood its consequences. The tools made thoughtlessness expensive.

Phototypesetting and the desktop revolution

The first phototypesetting machines appeared in the 1940s and became widespread through the 1950s and 60s. The principle was simple: instead

of casting metal, you projected light through a negative of each character onto photosensitive paper. The paper was then used to make printing plates. Metal, with its weight and heat and physical constraints, was gone.

This mattered for typography because it removed the physical limits that had governed type for five centuries. In metal, you could not easily overlap characters, because they were solid objects. In photo, you could. In metal, scaling a typeface required cutting new punches at every size. In photo, you turned a dial. The new freedom was real, but it was also dangerous: the optical corrections that type designers had built into metal fonts at each size — the way a 6-point typeface is subtly different in proportion from its 72-point version, adjusted for how the eye reads at different scales — were now routinely ignored. Designers scaled a single master up and down indiscriminately. The results were technically acceptable and often visually inferior.

The real rupture came in 1984 and 1985 in rapid succession. Apple released the Macintosh in January 1984. Adobe published the PostScript page description language in 1985. Apple released the LaserWriter printer, which contained a PostScript interpreter, also in 1985. Aldus released PageMaker, the first desktop publishing application, the same year.

PostScript was the pivot. It was a programming language whose output was a printed page — a way of describing, in mathematical terms, exactly where every shape and character on a page should appear. It meant that a computer could, for the first time, describe a page with the same precision as a professional typesetting system. The LaserWriter made that description physical. PageMaker gave ordinary users a graphical interface to drive it. And the Macintosh — with its then-revolutionary graphical interface and screen fonts that actually resembled what would print — put the whole system on a desktop that anyone could buy.

The desktop publishing revolution democratized access to type and layout. It also, immediately and catastrophically, democratized bad typography. Suddenly anyone with a Macintosh and a copy of PageMaker could produce a newsletter, a flyer, a business card, a book. Most of them could not. The result was a decade of documents set in too many

typefaces, at inconsistent sizes, with arbitrary spacing and no underlying grid, printed on laser printers and photocopied until the halftones fell apart. The tools arrived before the education. In some circles, this period is still spoken of with a shudder.

This is not a trivial point. It is the central problem this book exists to address. Access to powerful tools for arranging text on a page does not automatically produce good typography. It never has. The Linotype operator who set six lines a minute still needed to know about rivers, rags, widows, and the correct way to space small capitals. The PageMaker user who discovered they could apply seventeen fonts to a single paragraph needed — badly — to know why they shouldn't. The CLI typographer who can generate a PDF from Markdown in a single command needs to know enough about typography to make that PDF worth reading.

Tools are not enough. They are never enough. That is the lesson of 1985.

Knuth, TeX, and why it matters

In the 1970s, Donald Knuth was a computer scientist at Stanford working on a multi-volume work called *The Art of Computer Programming*, a monumental reference on algorithms that he had begun in 1962 and was still — decades later, volumes still incomplete — working on. In 1976, he received the galley proofs for the second edition of Volume 2. They had been set using the new phototypesetting systems that had displaced hot metal. He was appalled.

The mathematical notation — the core of the work, the thing that made it valuable — looked wrong. The spacing was inconsistent. The symbols were poorly positioned. The overall appearance was, by his judgment, substantially worse than the first edition, which had been set in hot metal. He sent the proofs back.

Then he did something that very few people, confronted with a tool that does not work to their satisfaction, actually do: he built a better one.

Knuth spent from 1977 to 1989 developing TeX (pronounced *tech*, from the Greek *τεχνή* — art, craft, skill). It was not a minor project undertaken on the side. He took a sabbatical from Stanford. He developed not just the typesetting system but a companion system called METAFONT for designing the letterforms themselves, and a family of typefaces called Computer Modern to go with it. He wrote three books documenting the work. The total effort was, by any measure, extraordinary — a decade of one of the finest computer scientists alive, dedicated to the problem of how mathematical text should look on a page.

What Knuth built was different from everything that had come before in a way that repays careful attention.

Previous typesetting systems — and most subsequent ones — set type *greedily*: they filled a line with as many words as would fit, then moved to the next line. TeX does not do this. TeX's line-breaking algorithm looks at the *entire paragraph at once*. It considers all the possible ways the paragraph could be broken into lines, assigns each a numerical “badness” score based on how stretched or compressed the word spacing would need to be, and finds the combination that minimizes total badness across all lines. A line that looks acceptable in isolation might be rejected because it forces a worse outcome two lines later.

This is a fundamentally different philosophy. It is the difference between local optimization — make each decision look reasonable in context — and global optimization — find the arrangement that produces the best result for the whole. It is why paragraphs set in TeX look different from paragraphs set in Word. Not always dramatically different, but consistently, cumulatively different in a way that the eye registers as quality without necessarily being able to name it.

TeX is also, famously, stable. Its version numbers converge to π : 3, 3.1, 3.14, 3.141, and so on. Knuth has specified that on his death, the version number should be set to π exactly and the program frozen forever. No new features, no updates, no security patches. The rationale is that TeX is used to typeset scientific and mathematical literature that must remain reproducible over centuries, and that instability in the typesetting system would be a disservice to that literature. A document set in TeX in 1985 should produce identical output in 2085. This is an unusual position in

software, where obsolescence is the default assumption, and it reflects a set of values — about permanence, about craft, about the responsibilities of toolmakers — that are worth sitting with.

In 1985, Leslie Lamport released LaTeX, a collection of macros built on top of TeX that made it dramatically more accessible to ordinary authors. Where TeX required you to understand its underlying mechanisms, LaTeX gave you higher-level commands: `\section{}` instead of formatting instructions, `\begin{itemize}` instead of manual list construction. LaTeX separated the concerns of content and presentation in a way that anticipated what CSS would later do for HTML. You wrote a document and specified its *structure*; a document class handled the appearance.

LaTeX became, and remains, the standard for scientific and mathematical publishing. If you submit a paper to a physics journal, an economics journal, or a computer science conference, you will almost certainly be submitting a LaTeX document. The major publishers — Springer, Elsevier, the ACM, the IEEE — maintain LaTeX document classes for their house styles. The majority of mathematical notation you read in academic papers was set by TeX.

It is also, by modern standards, difficult to use. The syntax is verbose. Error messages are opaque. Debugging a complex LaTeX document can require significant expertise. The learning curve is steep. These are real costs, and the book you are reading will not pretend otherwise. But understanding *why* TeX works the way it does — understanding the philosophy that Knuth built into it — will make you a better typographer regardless of which tool you ultimately use most. The questions TeX asks about text are the right questions.

The web and the fracturing of print

Tim Berners-Lee wrote the original proposal for the World Wide Web at CERN in 1989. The context matters: CERN was a large international physics laboratory, full of researchers who needed to share technical documents across different computers running different operating systems. The need was for something that worked everywhere, that required no

special software beyond a basic browser, and that could be read on screen without regard for how it might look printed. Precise typographic control was not on the requirements list.

HTML — Hypertext Markup Language — reflected these priorities. Its markup described the *structure* and *meaning* of a document (<h1> means “this is a first-level heading”; <p> means “this is a paragraph”) but said almost nothing about its appearance. What a heading looked like was up to the browser. What font a paragraph used was up to the browser. This was a feature, not a bug: it meant a document could be read on a text-only terminal, a graphical workstation, a phone, a screen reader for the visually impaired.

Early web typography was, by any classical standard, terrible. Documents were set in whatever fonts the user had installed — in practice, Times New Roman, Arial, and Courier on Windows; Times, Helvetica, and Courier on the Mac. There was no control over leading, no optical margin alignment, no microtypographic adjustment of any kind. The line-breaking was greedy in the crudest sense. Pages looked like typewriter output, formatted loosely.

Cascading Style Sheets, introduced in 1996 and gradually implemented (unevenly, incompletely, contentiously) over the following decade, began to restore typographic control to web documents. By the mid-2000s it was possible to set reasonable type on the web: to specify font sizes in relative units, to control line-height, to use font-variant for small capitals. Web fonts, via @font-face, became widely supported around 2009, which meant designers were no longer confined to system fonts. Google Fonts, launched in 2010, put hundreds of quality typefaces in the hands of any web developer at no cost.

By the 2010s, web typography had matured into a genuine discipline with its own literature, its own tooling, and its own aesthetics. But it had done so while preserving HTML’s foundational assumption: text *reflows*. Make the browser window narrower and the lines get shorter. Increase the user’s default font size and the whole layout shifts. The document does not have a fixed canvas. The designer proposes; the browser disposes.

This is the fundamental tension that runs through everything in this book. Print typography — the tradition that runs from Gutenberg through hot metal through TeX — assumes a fixed canvas. You know the page size. You know the margins. You know exactly how wide the text column is and exactly how many characters will fit on a line. Every typographic decision is made with full knowledge of the physical constraints.

Web typography assumes a variable canvas. The text column might be 400 pixels wide or 1200 pixels wide. The user might have changed the default font size. The device might not even have a screen in the usual sense. Typographic decisions must be robust to a range of conditions you cannot fully control or predict.

When you generate a PDF from Markdown using Typst or a LaTeX backend, you are working in the print tradition: fixed canvas, precise control, every detail in your hands. When you generate HTML from the same Markdown file, you are working in the web tradition: reflow, approximation, deference to the reader's environment. Both are legitimate. They are not the same.

The proliferation of formats that defines the current landscape — HTML, PDF, EPUB, DOCX, man pages, slides — is a direct consequence of this fracture. Each format embodies a set of assumptions about where and how the text will be read. EPUB is HTML that pretends, imperfectly, to be a book. DOCX is a format that tries to preserve some print-style layout while remaining editable. PDF is PostScript's descendant: a fixed-canvas, device-independent representation of a page, designed to look identical everywhere it is opened. Each has its place. None is universally best.

The CLI tools you will learn in this book — Pandoc, LaTeX, Typst, Quarto, and others — are largely tools for navigating this landscape: for taking a single source document and producing appropriate output for different contexts. The authoring format (usually Markdown) is neutral. The output formats embody specific assumptions. Your job, as a CLI typographer, is to understand those assumptions well enough to make the right choices and to apply enough typographic knowledge to each output to make the result worth reading.

That knowledge begins with the history. The units come from metal type. The concept of a document class comes from Knuth. The separation of content and presentation comes from both Lamport and Berners-Lee. The tension between fixed and reflow canvases comes from the collision of five centuries of print with three decades of the web. None of it is arbitrary. All of it is knowable.

That is what this book is for.

Typography Fundamentals

Typography is older than printing. The word comes from the Greek *typos* — impression, mark — and *graphia* — writing. Long before Gutenberg cast his first piece of type, scribes were making deliberate decisions about letterforms, spacing, and the arrangement of text on a page. What printing industrialized, and what the computer later digitized, was a body of knowledge and practice that had been accumulating for centuries.

This chapter covers that body of knowledge: the anatomy of letterforms, the classification of typefaces, the mechanics of spacing, and the principles that govern how text communicates. None of it is specific to any tool. LaTeX and CSS and Typst and Quarto all expose the same underlying concepts through different syntaxes. Learning the concepts once means you will recognize them everywhere.

A note on depth: typography is a field broad enough to fill entire careers. This chapter covers what you need to work intelligently with CLI typesetting tools — enough to make good decisions, understand why certain defaults exist, and diagnose problems when the output looks wrong. For those who want to go further, the further reading appendix lists the books that practitioners actually use.

Anatomy of a typeface

Every letter in a typeface is constructed from a set of parts that typographers have named. These names matter because they appear in technical documentation, in the specification of fonts, and in the conversations you will eventually have with designers or with yourself when something looks wrong and you need to articulate why.

The *baseline* is the invisible line on which letters sit. Most lowercase letters rest on it. Some — called *descenders* — dip below it: g, j, p, q, y. The parts that extend above the general height of lowercase letters are *ascenders*: the upward strokes of b, d, f, h, k, l, t. The height of a typical lowercase letter — specifically the flat-topped ones like x, n, a — is called the *x-height*. It is one of the most important measurements in typography because it strongly influences the apparent size and readability of a typeface. Two typefaces set at the same point size can look dramatically different in scale because their x-heights differ.

Above the x-height, ascenders reach toward the *cap height* — the top of capital letters — and sometimes beyond it. Below the baseline, descenders reach toward the *descender line*. The distance from the descender line to the cap height (or in some systems, to the top of the tallest ascender) is the overall *body height* of the type.



Figure 1: Anatomy of a typeface: baseline, x-height, cap height, ascenders, and descenders.

Individual letterforms are built from *strokes*. A stroke that ends in a tapered, wedge-shaped, or bracketed terminal is characteristic of *serif* typefaces; the serifs themselves are the small cross-strokes at the ends

of the main strokes. A typeface with no such terminals is *sans-serif* — without serif. The *stem* is the primary vertical stroke of a letter. The *bowl* is a curved stroke that encloses a counter: the round space inside a b, d, o, p, or q. The *counter* itself — that enclosed space — matters enormously to readability; as type gets smaller, counters close up first, which is why typefaces intended for small sizes are designed with more open counters.

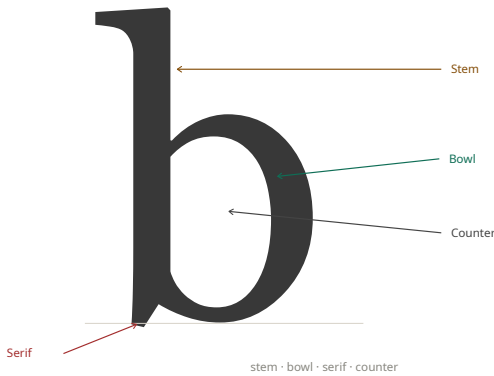


Figure 2: Strokes of a letterform: stem, bowl, serif, and counter.

Stress describes the axis of thick-to-thin variation in a letterform. In Renaissance typefaces, this axis is diagonal, echoing the angle at which a broad-nib pen is held. In neoclassical typefaces, it is vertical. In slab-serif faces, there is little variation at all. Stress is a useful diagnostic: when you are trying to identify or compare typefaces, the angle of stress is one of the first things to look at.

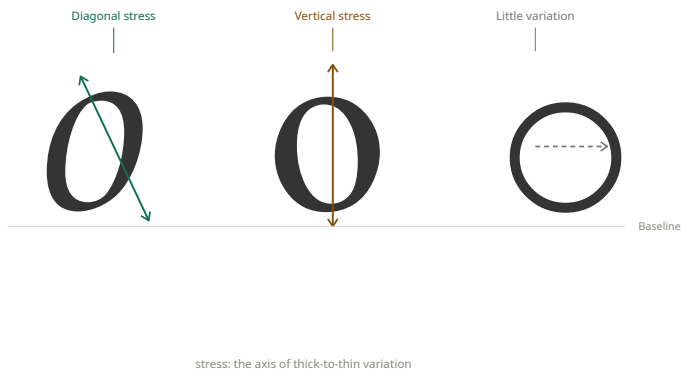


Figure 3: Stress: the axis of thick-to-thin variation.

Optical size is a concept that hot-metal type embodied and digital type initially abandoned. A typeface designed to be set at 8 points looks different from the same typeface designed for 72 points — the strokes are proportionally heavier, the counters more open, the letter spacing slightly wider. This is not because the type was scaled; it is because the designer drew it differently for each size, compensating for the way the eye reads at different scales. In metal, you had no choice but to do this: each size was a separate physical artifact. In early digital type, it was common to use a single master for all sizes, scaling it mathematically, which produced results that looked fine at display sizes and often looked weak or crowded in the text range. Variable fonts, introduced in 2016 as part of the OpenType specification, allow type designers to encode optical size as a continuous axis, among other variations. We will return to this in Chapter 3.

Classifying typefaces

Type classification systems are maps, and like all maps they simplify. The most commonly used scheme, developed by Maximilien Vox in 1954 and later adopted by the British Standards Institution, divides typefaces into a handful of broad families based on their historical origins and formal characteristics. Knowing these categories will not tell you which typeface to use — that judgment requires experience and context — but it will help you understand the choices available to you and communicate about them clearly.

Humanist typefaces trace their lineage to the letterforms of Italian Renaissance scribes working in the 15th century. Their stress is diagonal, their serifs bracketed (meaning the transition from stroke to serif is curved rather than abrupt), and their overall texture has an organic, slightly uneven quality that recalls the hand. Garamond, Bembo, and Palatino are canonical examples. They are warm, readable at text sizes, and carry associations of scholarship and tradition. In LaTeX, Palatino is available as a standard package option; it is an excellent choice for academic documents that want to move slightly away from the default Computer Modern.

Transitional typefaces emerged in the 18th century, as the influence of the pen decreased and the influence of the engraver's tool increased. Stress becomes more vertical, serifs become more refined, and the contrast between thick and thin strokes increases. Times New Roman is the most widely known example, designed in 1932 for the newspaper of the same name. Baskerville is the canonical 18th-century example. Transitional typefaces are neutral in a way that humanist faces are not: they carry less character, which makes them appropriate for contexts where the typography should not call attention to itself.

Didone or *modern* typefaces, developed in the late 18th and early 19th centuries by Firmin Didot and Giambattista Bodoni, took the transitional direction to its extreme. Stress is fully vertical, serifs are hairline-thin with no bracketing, and the contrast between thick and thin strokes is dramatic. They are beautiful at large sizes and in well-produced print. They

are difficult to read at small sizes on screen, where the hairline strokes disappear or render as a blur. Bodoni and Didot themselves are the canonical examples. Use them for display — headlines, covers, pull quotes — and rarely for body text.

Slab-serif typefaces replaced the hairline serifs of the Didone style with serifs of the same weight as the main strokes. They emerged in the early 19th century for advertising and display use, where legibility at distance mattered more than elegance at small size. Clarendon and Rockwell are classic examples. In the 20th century, slab serifs were rehabilitated for text use — Courier, the default monospace typeface of the typewriter era, is a slab serif — and designers like Herb Lubalin created refined slab-serif text faces. For CLI-produced documents, the slab serifs worth knowing are mostly in the monospace category, which brings us to the most practically relevant class for this book.

Monospace typefaces give every character the same horizontal width. A lowercase i and a capital M occupy the same amount of space. This was a constraint of the mechanical typewriter, where a fixed-width platen advance meant variable-width type was impractical. It survived into computing because early terminals could only display fixed-width character grids. Today, monospace typefaces are used primarily for code, terminal output, and anything that needs to appear as if it came from a machine. In typeset documents, they should be used sparingly and consistently — for code examples, command-line invocations, and file paths. The default monospace faces in most systems (Courier, Courier New) are functional but plain; for serious work, typefaces like Inconsolata, JetBrains Mono, or IBM Plex Mono offer considerably more refinement.

Sans-serif typefaces, absent the serifs entirely, first appeared in the early 19th century and were initially considered crude and utilitarian. Their rehabilitation came largely through the Bauhaus school in 1920s Germany and the Swiss typographic movement of the 1950s, which produced the International Style and typefaces like Helvetica and Univers. Sans-serif faces subdivide further: *grotesque* sans-serifs (Helvetica, Akzidenz Grotesk) have roots in the 19th century and a slight irregularity that registers as warmth; *geometric* sans-serifs (Futura, Avenir) are constructed

from circles and straight lines, rational and impersonal; *humanist* sans-serifs (Gill Sans, Frutiger, Myriad) borrow their proportions from roman letterforms and tend to have the best readability at text sizes. For body text in a typeset document, humanist sans-serifs perform best among the sans categories; for headers and display, all three types have legitimate applications.

Display typefaces are designed for use at large sizes — above approximately 24 points — and are not suitable for body text. They include decorative faces, script faces, and extremely high-contrast or eccentric designs that become illegible when reduced.

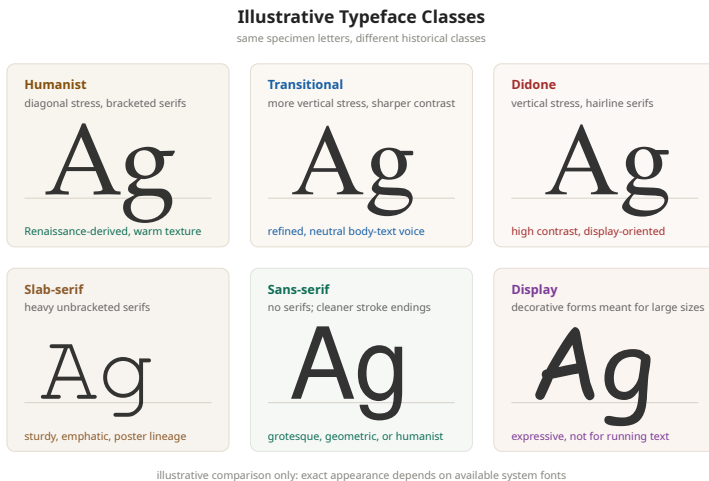


Figure 4: Illustrative comparison of humanist, transitional, didone, slab-serif, sans-serif, and display typeface classes.

For the practical work in this book, you will most often be working with serif faces for body text in print-oriented documents, humanist sans-serif faces for screen-oriented documents, and monospace faces for code. The document examples in Part IV will specify typefaces by name and explain the reasoning behind each choice.

Spacing: kerning, tracking, and leading

If typeface selection is the choice of which letter shapes to use, spacing is the decision about how to arrange them relative to each other. Spacing determines whether text feels comfortable or cramped, airy or dense. It is the dimension of typography that most distinguishes professional work from amateur work, because it requires attention that software does not provide automatically.

Tracking (also called *letter-spacing*) is the uniform adjustment of space between all characters in a range of text. Increasing tracking opens the text up; reducing it tightens it. The appropriate amount of tracking varies with the size of the type: display-sized type (above 24 points or so) generally benefits from slightly tighter tracking than the default, because at large sizes the spaces between letters begin to read as gaps rather than intervals; body text should be set at default tracking or very slightly loose; very small text (below 9 points) benefits from slightly open tracking to improve legibility.

In LaTeX, tracking is controlled by the `microtype` package, which we will discuss in Chapter 26 under microtypography. In CSS, it is the `letter-spacing` property. In Typst, it is the `tracking` parameter. In all cases, the unit is a fraction of an em, and adjustments should be measured in small increments — the difference between good tracking and poor tracking is often less than 2% of an em.

Kerning is the adjustment of space between specific pairs of letters. Most fonts include kerning tables — lists of pairs that need special treatment because their shapes create awkward gaps or collisions. The classic example is the pair AV: set in a typeface with straight-sided A and V, the two letters create a visible hole between them that looks like too much space, even if the inter-character spacing is technically uniform. A kern brings them closer. Other common kerning pairs include To, Ty, We, and fa. Well-made fonts include hundreds or thousands of such pairs.

The distinction between tracking and kerning matters because they solve different problems. Tracking adjusts all spacing globally; kerning adjusts specific pairs. In practice, you should rely on the kerning built into your

typeface and adjust it only rarely. What you should ensure is that kerning is turned on — some applications and pipelines disable it by default, especially in small print runs or quick previews, because it is computationally expensive. In LaTeX, kerning is enabled by default in the traditional engines and requires `microtype` for additional refinement. In CSS, font kerning can be enabled with `font-kerning: normal`.

Leading (pronounced *ledding*, from the lead strips of the hot-metal era) is the vertical distance from the baseline of one line to the baseline of the next. In digital typography, this concept is expressed as *line-height* in CSS and as `\baselineskip` in LaTeX. The relationship between leading and type size determines how dense or airy the text feels. A line-height of 1.0 (equal to the type size) means lines sit directly against each other with no breathing room — practically unreadable for body text. A line-height of 2.0 creates generous space but can fragment the text into islands that the eye struggles to connect. For body text in most contexts, a line-height of 1.4 to 1.6 times the type size is appropriate; wider columns benefit from slightly more leading, narrower columns from slightly less.

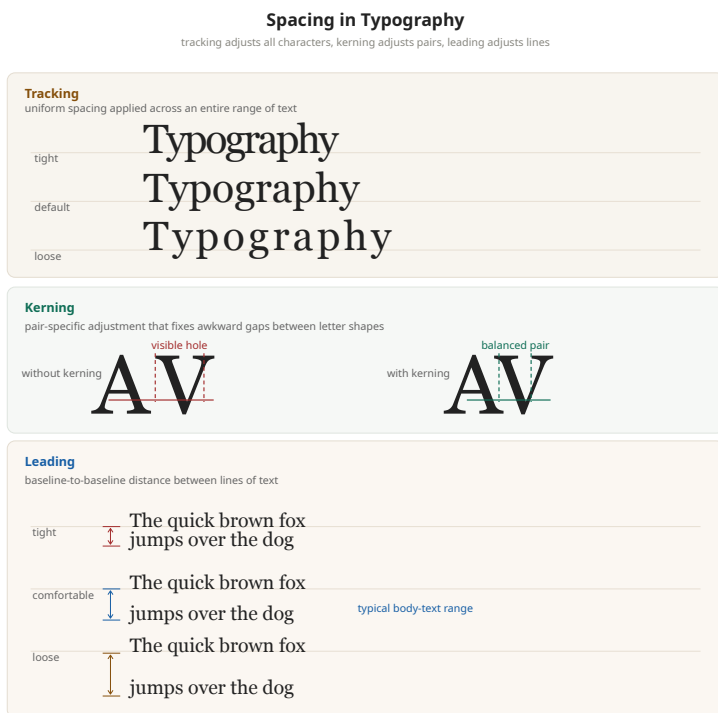


Figure 5: Illustration of tracking across a word, kerning in the AV pair, and leading between baselines.

The relationship between type size, line length, and leading forms an interconnected system. The reader's eye needs to be able to track from the end of one line to the beginning of the next without losing its place. A long line makes this journey more difficult and requires more leading to compensate. A short line can survive with less leading. Robert Bringhurst, in *The Elements of Typographic Style* — the book that professional typographers most often cite as essential — states this as a principle: for a measure of approximately 65 characters, a leading of 1.2× to 1.4× is appropriate; for a measure of 80 to 90 characters, 1.4× to 1.6× is better.

What is the right line length? For Latin-script body text in print, the conventional wisdom — derived from reading research and long typographic practice — is 45 to 75 characters per line, with 65 as the oft-cited ideal. Shorter than 45 and the eye makes too many returns, breaking the reading rhythm; longer than 75 and the eye begins to lose its place at the line return. These are not hard limits; they are regions of comfort. Long-form reading benefits from being within them. Short-form text — captions, footnotes, tables of contents — can be set at shorter measures without harm.

In LaTeX, line length is determined by the `\textwidth` parameter, which is set by the document class and page geometry. In CSS, it is most reliably controlled with the `max-width` or `width` property on the body element or text container, and the `ch` unit (the width of the character “o” in the current font) is particularly useful: `max-width: 65ch` enforces a roughly 65-character measure regardless of font size.

The typographic scale

Type is not set at arbitrary sizes. Professional typography uses a *type scale* — a set of related sizes that form a visual hierarchy. The sizes in a scale are not chosen at random; they stand in relationships to each other that make the difference between adjacent levels perceptible without being jarring.

The oldest type scale in continuous use is based on the sizes standardized by European printers over several centuries: 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 21, 24, 36, 48, 60, 72. These sizes appear in the default size options of LaTeX, in the presets of many desktop publishing applications, and in the suggestions of most typographic guides. They are not mathematically rigorous, but they work: the intervals between them feel proportional because they roughly approximate a geometric progression.

A more systematic approach uses the *modular scale*, popularized by Tim Brown and ultimately derived from Bringhurst: choose a base size (typically your body text size, often 10pt or 12pt) and a ratio (often the golden ratio, 1.618, or simpler ratios like 1.333 or 1.5), and generate each scale step

by multiplying by the ratio. A scale based on 12pt and a ratio of 1.333 produces: 12, 16, 21.3, 28.4, 37.9, 50.5. These become your body, subheading, heading, section, display, and hero sizes.

In practice, the exact scale matters less than the consistency. A document that uses three or four sizes, applied consistently to a clear hierarchy — body text, captions, headings, title — will read more professionally than a document that uses eight sizes applied loosely. The common failure mode is proliferation: using a slightly different size for every level of heading, for captions vs. footnotes vs. callouts, until the document has no hierarchy at all but only a cloud of competing sizes.

LaTeX document classes enforce a typographic scale by defining a small set of size commands (`\tiny`, `\scriptsize`, `\footnotesize`, `\small`, `\normalsize`, `\large`, `\Large`, `\LARGE`, `\huge`, `\Huge`) as multiples of the document's base font size. This is good design. The discipline it imposes — you should have a reason to use a non-standard size, and that reason should be strong — is worth preserving even when working in systems that allow arbitrary sizes.

Hierarchy, contrast, and the reader's eye

A typeset document is not a flat surface. It has depth — levels of importance, structure that the reader needs to navigate, signals that distinguish one kind of content from another. Typography is the visual language through which that depth is expressed. *Hierarchy* is the term for the system of visual distinctions that lets a reader see, at a glance, what a document contains and how its parts relate to each other.

Hierarchy is created through contrast. Elements at different levels of importance should look different from each other, and the difference should be perceptible immediately, without requiring the reader to parse the text to understand what they are looking at. A chapter title should look like a chapter title. A section heading should look subordinate to a chapter title. A caption should look subordinate to body text. Body text should look comfortable and neutral. Nothing should look important unless it is.

Contrast can be created through several channels: *size* (larger elements read as more important), *weight* (heavier elements — bolder type — read as more prominent), *style* (italic or small-caps type differs from roman), *color* or *value* (darker or more saturated elements command attention), *spacing* (more surrounding white space emphasizes an element), and *position* (elements at the top of a hierarchy tend to sit at the top or left of the page).

The failure mode on one side is insufficient contrast: a document where headings are only slightly larger or slightly bolder than body text, where the hierarchy exists in the markup but not on the page. The reader's eye does not know where to start; navigating the document requires effort. The failure mode on the other side is excessive contrast: too many levels, too many weights, too many sizes, a page that shouts instead of speaking. The reader's eye is pulled in too many directions at once and gives up.

Good hierarchy typically involves three to four levels at most: a title or chapter-level element, one or two levels of heading, and body text. Captions, footnotes, and marginalia form their own sub-hierarchy, consistently distinct from the main text but clearly subordinate to it. Beyond four levels, distinctions become difficult to perceive reliably and the hierarchy tends to collapse.

In LaTeX, the standard document classes provide a ready-made hierarchy of `\part`, `\chapter`, `\section`, `\subsection`, `\subsubsection`, and `\paragraph`. This is probably one level too many for most documents: the difference between a `\subsubsection` and a `\paragraph` is rarely perceptible, and documents that use all six levels are usually documents with structural problems that typography cannot solve. In CSS, you have six levels of heading (h1 through h6) and the same caveat applies.

Emphasis is a special case of contrast — the temporary departure from the neutral appearance of body text to signal importance. Italic is the conventional form of emphasis in Latin-script text. Bold is a stronger form, appropriate for terms being defined, warnings, or labels. Both should be used sparingly: emphasis only works when it is exceptional. A paragraph with eight words in bold and five phrases in italic has no emphasis at all; it has noise.

Small capitals — letterforms with the shape of capitals but sized to match the x-height of lowercase — are the correct form for abbreviations (AD, BC, CEO, ISBN) set within body text. Full-size capitals in running text look too large and create interruptions in the reading rhythm. Most professional typefaces include a true small capitals variant, cut with strokes proportionally heavier than a simply scaled-down capital. LaTeX enables small capitals with `\textsc{}` when using a typeface that includes them. CSS uses `font-variant: small-caps`. If your typeface does not include true small capitals and the software fakes them by scaling regular capitals down, the result will look thin and slightly wrong — a good reason to choose typefaces that include the full range of OpenType features.

The reader's eye, moving through a well-set page, should encounter no surprises. Not visual surprises — not sudden changes in size or weight that were not part of the established hierarchy — and not cognitive surprises — not ambiguity about whether something is a heading or a caption or an emphasis or a pull quote. The job of typography is to be invisible: to make the structure of the content so clear that the reader can follow it without thinking about the medium. When typography succeeds completely, the reader remembers what they read, not how it looked.

That is, in the end, the standard. Not beauty, though good typography is often beautiful. Not complexity, though complex documents require complex typographic solutions. The standard is: does the reader get what the author intended, without friction, without confusion, and without the medium inserting itself between the content and the mind?

Everything in this book is in the service of that standard.

Digital Typesetting Concepts

The transition from metal type to digital type was not simply a change of material. It was a change of *model*. A piece of metal type is a physical object with fixed dimensions. A digital typeface is a set of mathematical descriptions — outlines, metrics, instructions — that a rendering engine interprets to produce marks on a surface. The surface might be a screen, a laser printer, an inkjet printer, or an imagesetter producing film for a printing press. The same description serves all of them.

This abstraction is powerful and introduces its own complexities. When type existed only as metal, the relationship between the type and its output was direct: you pressed metal against paper and got an impression. In digital typography, there is a chain of interpretation between the font file and the marks on the page, and every link in that chain involves decisions — about resolution, about rendering, about color, about encoding — that affect the final result. Understanding that chain is what this chapter is about.

Units of measurement

Typography inherited a patchwork of measurement units from centuries of print practice in different countries, and the digital era added several more. CLI typesetting tools use most of these units, and confusion between them is one of the most common sources of errors in document configuration.

The *point* is the fundamental unit of typographic measurement. Its precise value has varied historically, but the desktop publishing point — standardized by Adobe and now universal in digital systems — is exactly $1/72$ of an inch, or approximately 0.353 millimeters. When you specify a

font size of 12 points, you are specifying that the em square (the invisible box that contains the character, including its ascenders, descenders, and side bearings) is $12/72$ of an inch tall. Note that this does not mean any particular part of the letter is 12 points tall: the relationship between the em square and the visible letterforms is determined by the type designer and varies between typefaces.

The *pica* is 12 points, or $1/6$ of an inch. It is used primarily for specifying larger dimensions — column widths, margins, and spacing — in print layout. In LaTeX, picas are abbreviated as pc. In CSS, they are available as pc as well, though rarely used.

The *em* is a relative unit equal to the current font size. If you are setting 12-point type, one em is 12 points. If you change to 10-point type, one em becomes 10 points. This relativity is what makes the em useful: measurements expressed in ems scale automatically when the font size changes. In LaTeX, the em is used constantly — `\hspace{1em}`, `\setlength{\parindent}{1.5em}` — and in CSS it is one of the primary units for spacing and sizing. The *en* is half an em. Both units are used for spacing: an em dash (—) is nominally one em wide; an en dash (–) is nominally one en.

The *rem* (root em) is a CSS-specific unit equal to the font size of the root element of the document, typically the `html` element. Where the em scales with local font size, the rem is constant across the document. This makes it useful for establishing a consistent baseline grid: set a base font size on `html` and express all spacing in rems, and the proportions hold everywhere. Many modern CSS typographic systems use a combination of rems for global structure and ems for component-level spacing.

The *ex* is the x-height of the current font — the height of lowercase letters without ascenders. It is less commonly used than the em but appears in some LaTeX spacing commands and in CSS, where `1ex` scales with the optical size of the type rather than its nominal size.

Absolute units — *millimeters* (mm), *centimeters* (cm), and *inches* (in) — are useful for specifying page dimensions, margins, and other physical measurements in print-oriented documents. In LaTeX, page geometry is typically specified in absolute units using the `geometry` package:

```
\usepackage[a4paper, margin=25mm]{geometry}
```

In CSS, absolute units are meaningful only for print stylesheets (@media print); on screen, a CSS inch is 96 device-independent pixels, which may not correspond to a physical inch depending on the display's actual pixel density.

The *pixel* (px in CSS) is a screen unit with no direct equivalent in print systems. In CSS, it is a device-independent unit equal to 1/96 of a CSS inch — not necessarily one physical pixel on the display, but a unit that the browser scales appropriately for the device. Pixels are common in CSS typography but should be avoided for font sizes in favor of relative units (em or rem), because absolute pixel sizes do not respect the user's browser font size preferences, which is both an accessibility issue and, in many jurisdictions, a legal compliance concern.

The practical rule for CLI typesetting is: use points and picas for print documents in LaTeX or Typst; use ems and rems for web output in CSS; use absolute units (mm or in) for page geometry; and avoid pixels for font sizes.

Font formats

A font file is not simply a collection of pictures of letters. It is a structured data format containing, at minimum, the outlines of the characters, the metrics that govern their spacing, and the metadata that identifies the font. Modern font formats contain considerably more than this minimum.

TrueType (.ttf) was developed jointly by Apple and Microsoft in the late 1980s as a response to Adobe's Type 1 format, which required a separate rasterizer. TrueType fonts contain character outlines described as quadratic Bézier curves and include *hinting* instructions — hand-coded or automatically generated rules that adjust the rendering of outlines at small sizes and low resolutions to improve legibility. TrueType remains widely used, particularly on Windows and in web contexts.

OpenType (.otf) was developed in the mid-1990s as a collaboration between Microsoft and Adobe and officially launched in 2000. It subsumes both TrueType and Type 1 outline formats within a single container and, more importantly, introduces an extensive set of *features* that go far beyond basic character rendering: ligatures, small capitals, old-style figures, swash characters, contextual alternates, stylistic sets, and language-specific forms. An OpenType font with a full feature set is a substantially more capable object than its predecessors. In LaTeX, OpenType features are available through XeLaTeX and LuaLaTeX (not through the original pdfLaTeX engine); in CSS, they are accessible via the `font-feature-settings` property.

WOFF and *WOFF2* (Web Open Font Format) are web-specific wrappers around TrueType or OpenType fonts, with compression applied. WOFF2, introduced in 2018, offers significantly better compression than WOFF and is the recommended format for web fonts today. WOFF fonts are not used in print typesetting pipelines; they are served by web servers and loaded by browsers.

Variable fonts, introduced as part of the OpenType specification in 2016, represent a significant departure from the traditional model of a font family as a collection of separate files — one per weight, one per width, one per style. A variable font encodes the entire design space of the family in a single file, with one or more *variation axes* (commonly weight, width, optical size, and italic) along which the design continuously varies. A variable font with a weight axis might allow you to set type at any weight from 100 to 900, not just at the discrete stops of Thin, Light, Regular, Medium, Bold, ExtraBold, and Black. In CSS, variable fonts are controlled with `font-variation-settings` and the standard properties (`font-weight`, `font-stretch`, `font-style`). In print workflows, variable font support is more limited but growing.

For CLI typesetting, the practical implications of font formats are:

- **pdfLaTeX** can only use fonts in its own internal format (Type 1 or specially prepared packages). The standard LaTeX font packages — `palatino`, `times`, `helvet`, `courier`, and many others — handle

this conversion transparently. You do not work with font files directly in pdfLaTeX.

- **XeLaTeX and LuaLaTeX** can use any OpenType or TrueType font installed on your system, addressed by name. This is the path to using modern professional typefaces in LaTeX documents.
- **Typst** uses OpenType and TrueType fonts directly, addressed by file path or family name.
- **Pandoc HTML output** uses CSS font stacks and web fonts, loaded via @font-face or Google Fonts links in a custom template.

The choice of LaTeX engine — pdfLaTeX vs. XeLaTeX vs. LuaLaTeX — is largely a choice about font access. pdfLaTeX is faster and more portable; the other two engines open up the full range of system fonts and OpenType features. For documents that can use standard LaTeX font packages, pdfLaTeX is often the right choice. For documents that require specific professional typefaces, Unicode support beyond basic Latin, or advanced OpenType features, XeLaTeX or LuaLaTeX is necessary.

Rasterization and hinting

A font outline is a mathematical description of a letter's shape, defined by curves and straight lines. A display or printer is a grid of discrete dots — pixels or ink droplets. The process of converting the continuous outline into marks on a discrete grid is called *rasterization*.

At large sizes, rasterization is straightforward: the grid is fine enough that the outline can be followed closely, and the result looks smooth. At small sizes, rasterization becomes difficult: the grid is coarse relative to the feature size of the letter, and small decisions about which pixels to fill can dramatically affect legibility. The stem of a lowercase letter at 8 points on a 96dpi screen might be only one or two pixels wide. Whether those pixels are filled fully or partially, and how anti-aliasing softens the edges, determines whether the type is readable.

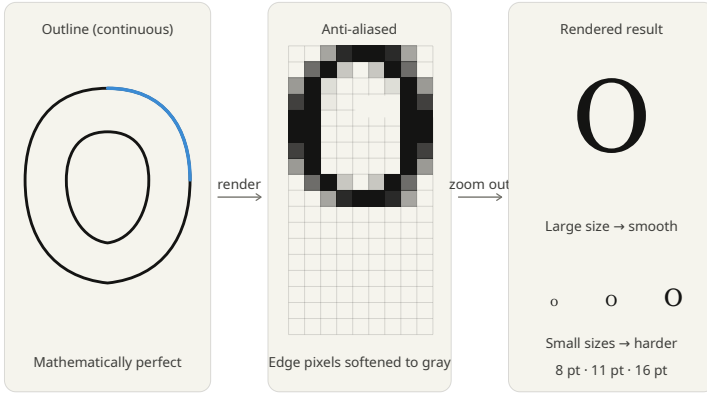


Figure 1: The journey from a mathematically perfect curve (left) to a pixel grid where edge squares are filled at fractional gray values to approximate the outline (centre), producing results that look smooth at large sizes but degrade to a handful of ambiguous pixels at small ones (right).

Hinting is the mechanism by which type designers (or font-generation tools) encode instructions to guide the rasterizer at small sizes. Hints tell the rasterizer: snap this stem to the nearest pixel grid, align these three bowls to the same height, maintain this minimum stroke width even when the outline would suggest thinner. Good hinting is laborious to produce by hand and requires deep expertise. It is one reason that high-quality fonts from professional foundries differ from free or automatically generated fonts, particularly at screen sizes below 16px.

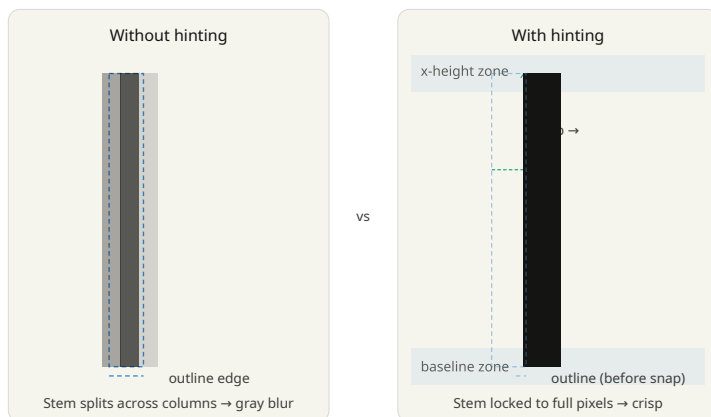


Figure 2: Hinting snaps a stem that would straddle pixel columns (left, blurred across three gray columns) to the nearest clean boundary (right, two fully black columns).

Modern screen rendering systems use *subpixel rendering* and *anti-aliasing* to improve apparent resolution. ClearType on Windows and CoreText on macOS both use knowledge of the physical layout of LCD subpixels (each physical pixel is composed of red, green, and blue subpixels arranged horizontally) to provide effective horizontal resolution three times higher than the pixel count would suggest. Anti-aliasing softens the stepped edges of letterforms by setting edge pixels to intermediate gray values rather than full black or full white.

For print output — laser printers at 300–600dpi, imagesetters at 1200–2400dpi — rasterization is much less of a concern. At these resolutions, outlines render cleanly at any reasonable body text size. The typographic quality of print output is determined primarily by the quality of the typeface design, the line-breaking algorithm, and the spacing decisions, not by rendering artifacts.

Understanding rasterization matters for CLI typesetting primarily in two contexts: when generating HTML output intended for screen reading, where font choice and size should be tested at actual screen resolution; and when generating PDFs intended for on-screen reading rather than print, where the same concerns apply.

Print versus screen

The difference between print and screen typography is not merely aesthetic. It is a difference in fundamental constraints that propagates through every typographic decision.

Resolution is the most obvious difference. A high-quality laser printer outputs at 600 dots per inch. An offset printing press can produce halftones at 150 lines per inch with effective resolution considerably higher. A modern high-density laptop display (the Retina-class displays Apple introduced in 2012, and their equivalents from other manufacturers) renders at approximately 200–250 pixels per inch in hardware, though CSS typically treats this as 96–192 logical pixels per inch depending on the device pixel ratio. A standard desktop monitor is 96–110 pixels per inch. The gap between print and screen resolution, which was enormous in the 1980s and 1990s, has narrowed substantially but has not closed.

The practical consequence for typography is that fine details — hairline serifs, the thin strokes of Didone typefaces, the delicate optical sizing adjustments in high-quality fonts — are visible in print and may not be on screen. Type that is beautiful in a PDF may look heavy or blurry when rendered on a moderate-resolution display. This argues for testing HTML output in a browser, not just inspecting a PDF, and for choosing typefaces appropriate to the output medium.

Color in print is modeled in CMYK — Cyan, Magenta, Yellow, and Key (black) — the four inks used in offset printing. Color on screen is modeled in RGB — Red, Green, Blue — the three channels of light used by displays. The conversion between these color spaces is not lossless: some colors that look vivid on screen cannot be reproduced in CMYK ink, and

some colors achievable in print look different on screen. For documents that contain color beyond black text — diagrams, photographs, decorative elements — awareness of the target color space is important.

For documents whose primary output is PDF intended for professional printing, color should be managed in CMYK from the start. LaTeX with the `colorspace` package and appropriate options can produce CMYK PDF output. The specific requirements for print-ready PDF — color space, resolution, fonts embedded or outlined, bleed and crop marks — are determined by the printer or print service, and we will cover the standard requirements in Chapter 8.

For documents whose primary output is HTML or screen-optimized PDF, RGB color is appropriate and CMYK is irrelevant. The concern is instead the consistency of color across different displays, which vary in color calibration, brightness, and gamut. Using standard web colors from an established palette rather than arbitrary hex values reduces the risk of colors looking significantly different on different devices.

Bleed is a print-specific concept. When a design element — a background color, a photograph, a colored border — is intended to run to the very edge of the printed page, it must actually extend *beyond* the intended trim edge of the page, because cutting is not perfectly precise. The standard bleed is 3mm or 1/8 inch beyond the trim edge on all sides. The document is printed on a slightly oversized sheet and then cut down; the bleed ensures that slight inaccuracies in cutting do not leave a white edge where the colored element was supposed to reach the page edge.

In LaTeX, bleed can be added with the `geometry` package by specifying a slightly larger page size and positioning the content appropriately. Most commercial printing workflows expect PDF files with bleed areas marked by crop marks. The `cropmarks` and related packages provide this in LaTeX.

For CLI-produced documents, bleed is relevant primarily for covers, promotional materials, and documents with full-bleed design elements. Standard academic papers, technical reports, and books with white backgrounds and standard margins do not require bleed.

Unicode, encoding, and OpenType features

Until the 1990s, most computer text encoding systems were limited to a small set of characters — typically 128 or 256 — adequate for one language but not for many. ASCII, the American Standard Code for Information Interchange, encoded 128 characters: the Latin alphabet in upper and lower case, the digits, punctuation, and control characters. Extended ASCII variants added another 128 characters, enough for Western European accented characters but nothing for Greek, Cyrillic, Arabic, Chinese, Japanese, or the thousands of other scripts used in the world's writing systems.

Unicode, first published in 1991 and now in its fifteenth major version, solves this problem by assigning a unique number — a *code point* — to every character in every writing system that human beings have used, living or dead. Unicode currently encodes over 149,000 characters. Every Arabic letter, every Chinese ideograph, every mathematical symbol, every emoji, every hieroglyph is in there.

UTF-8 is the encoding form of Unicode most relevant to CLI typesetting. It represents code points as sequences of one to four bytes, with the first 128 code points (the ASCII characters) encoded as single bytes identical to ASCII. This backward compatibility means that an ASCII text file is also a valid UTF-8 file. UTF-8 is the standard encoding for HTML files, Markdown files, and most modern plain-text formats. It is the encoding you should use for all source documents. In LaTeX, you declare UTF-8 input encoding with `\usepackage[utf8]{inputenc}` when using pdfLaTeX; XeLaTeX and LuaLaTeX assume UTF-8 by default.

The practical consequences for typesetting are significant. With UTF-8 input and appropriate font support, you can include accented characters, ligatures, mathematical notation, and non-Latin scripts directly in your source document as literal characters rather than as escaped sequences or special commands. This naïve café is more readable in source and less error-prone than `na\{i}ve caf\'{e}`. Whether your toolchain supports this transparently depends on the engine and font.

Ligatures are single glyphs that replace two or more adjacent characters whose shapes would otherwise conflict or create awkward spacing. The most common in Latin typography are *fi*, *fl*, *ff*, *ffi*, and *ffl* — combinations where the hook of the *f* would collide with the dot of the *i* or the ascender of the *l*. Quality typefaces include these standard ligatures and many more. In pdfLaTeX with the `fontenc` and `inputenc` packages configured correctly, standard ligatures are formed automatically. In XeLaTeX and LuaLaTeX, ligature behavior is controlled by OpenType feature settings.

Beyond ligatures, OpenType fonts may include a rich set of alternate character forms accessible through feature tags:

- `onum` — *old-style figures* (also called text figures): numerals with ascenders and descenders (1, 2, 3...) that sit more comfortably within lowercase text than the default *lining figures* (which are the same height as capital letters).
- `smcp` — *small capitals*: true small-capital forms for all uppercase letters.
- `sup` and `sub` — *superscript* and *subscript* forms: properly drawn raised and lowered characters, preferable to mathematically scaled versions for footnote markers, ordinals, and chemical formulas.
- `frac` — *fractions*: properly formed diagonal fractions ($\frac{1}{2}$, $\frac{3}{4}$) rather than the $1/2$ and $3/4$ that result from putting a slash between two full-size digits.
- `kern` — *kerning*: as discussed in Chapter 2, this enables the font's built-in pair-specific spacing adjustments.
- `liga` — *standard ligatures*: the basic set described above.
- `dlig` — *discretionary ligatures*: additional ligatures (*ct*, *st*, *sp*, and others) that are a matter of stylistic choice rather than necessity.
- `calt` — *contextual alternates*: character forms that change based on adjacent characters, common in script typefaces and some italic designs.

In LaTeX, OpenType features are accessed through the `fontspec` package (which requires XeLaTeX or LuaLaTeX):

```
\usepackage{fontspec}
\setmainfont{EB Garamond}[
  Numbers = OldStyle,
  Ligatures = Discretionary,
]
```

In CSS, they are accessed through `font-feature-settings`:

```
body {
  font-family: 'EB Garamond', serif;
  font-feature-settings: "onum" 1, "liga" 1, "dlig" 1;
}
```

Not all fonts support all features. A font that claims to be an OpenType font but includes only the basic Latin character set and no features is technically compliant but practically limited. When evaluating a typeface for use in a serious document, checking the breadth of its character coverage and OpenType feature support is worthwhile. The FontForge application and the `fc-query` command-line tool can both report a font's features; we will look at these in Chapter 4.

The concepts in this chapter — units, formats, rendering, color models, encoding, and features — form the technical substrate of everything that follows. They are not the most glamorous aspects of typesetting, but they are the ones that bite you: the PDF with unembedded fonts that the printer refuses to accept, the HTML that ignores the user's font size preference, the LaTeX source with a smartly-quoted character that pdfLaTeX cannot process, the ligatures that disappear when you switch PDF engines. Understanding the substrate means understanding why these failures happen and how to prevent them.

The next chapter brings this technical knowledge to bear on a practical task: managing fonts from the command line.

Font Management from the CLI

Fonts are software. They are files, stored in directories, loaded by programs, updated by package managers, and subject to licenses. Most operating systems provide graphical tools for managing them — font books, preview panels, installation wizards — but every one of those operations has a command-line equivalent, and for the CLI typographer, the command-line path is faster, scriptable, and often more informative.

This chapter covers font management on Linux, which is the natural home of CLI typesetting workflows. The tools described here — the `fontconfig` suite, primarily — are also available on macOS, and the concepts transfer to Windows with some adaptation. If you are running a headless server, a CI/CD pipeline, or a Docker container that builds documents, this chapter is especially relevant: there is no font book, no graphical installer, no preview panel. There is only the command line and, if you are lucky, a clear error message when the font you need is not found.

How Linux finds fonts: `fontconfig`

On Linux, font discovery and configuration is handled by `fontconfig`, a library and set of utilities written by Keith Packard and first released in 2001. `Fontconfig` does three things: it maintains a catalog of all fonts installed on the system, it provides a query interface for finding fonts by name or property, and it implements a substitution and matching algorithm that resolves requests for named fonts to actual font files.

Every application that renders text on Linux — whether a GUI application like a web browser or a CLI application like a LaTeX engine — typically uses `fontconfig` (directly or through a higher-level library like

Pango or FreeType) to find font files. When your LaTeX document requests the typeface “EB Garamond” and the engine finds the right .otf file on your system, that is fontconfig at work.

The fontconfig catalog is built from font files in a set of standard directories:

- `/usr/share/fonts/` — system-wide fonts installed by package managers
- `/usr/local/share/fonts/` — system-wide fonts installed manually
- `~/.local/share/fonts/` — per-user fonts (preferred on modern systems)
- `~/.fonts/` — per-user fonts (legacy location, still supported)

When you install a font package via `apt`, `dnf`, or another package manager, the font files land in `/usr/share/fonts/` and fontconfig picks them up automatically. When you install a font manually — by downloading from Google Fonts or purchasing from a type foundry — you copy the files to one of the per-user directories and rebuild the cache.

The fontconfig configuration lives in `/etc/fonts/fonts.conf` and the directory `/etc/fonts/conf.d/`. The main configuration file specifies which directories to scan, how to handle font substitution, and rendering hints for anti-aliasing and subpixel rendering. You should not need to edit the main configuration file; local customizations go in `/etc/fonts/local.conf` or in per-user configuration at `~/.config/fontconfig/fonts.conf`.

The font cache — fontconfig’s index of installed fonts — is stored in `~/.cache/fontconfig/`. It is rebuilt automatically when fonts are added or removed, but you can force a rebuild manually:

```
fc-cache -fv
```

The `-f` flag forces a rebuild even if fontconfig thinks the cache is current; `-v` (verbose) prints each directory as it is scanned, which is useful for confirming that your newly installed font directory was found. After

installing fonts manually, always run `fc-cache -f` before trying to use them.

fc-list: surveying what you have

`fc-list` prints a list of all fonts known to `fontconfig`. With no arguments, it produces one line per font face — typically hundreds or thousands of lines on a system with a reasonable font collection:

```
fc-list
```

```
/usr/share/fonts/truetype/dejavu/DejaVuSerif-Bold.ttf: DejaVu
↳ Serif:style=Bold
/usr/share/fonts/truetype/google-fonts/Poppins-Bold.ttf:
↳ Poppins:style=Bold
/usr/share/fonts/truetype/freefont/FreeSansOblique.ttf:
↳ FreeSans:style=Oblique
...
```

The output format is `path: family:style=style`. This raw output is most useful when piped through `grep` or `sort`:

```
# Find all fonts whose name contains "Garamond"
fc-list | grep -i garamond

# List all installed font families, sorted, without duplicates
fc-list : family | sort -u

# Count total installed font faces
fc-list | wc -l
```

`fc-list` accepts a *pattern* as its first argument to filter results, and a list of *properties* to include in the output. The pattern uses `fontconfig`'s matching syntax: a colon-separated list of `property:value` pairs. Some useful filters:

```
# List only monospace fonts
fc-list :spacing=100 family

# List only bold fonts
fc-list :weight=200 family

# List fonts with support for Greek script
fc-list :lang=el family

# List fonts with support for Arabic
fc-list :lang=ar family
```

The spacing values are fontconfig constants: 0 is proportional, 90 is dual-width (some CJK fonts), 100 is monospace, 110 is charcell (old-style terminal fonts). Weight values follow a 0–210 scale where 80 is Regular, 100 is Medium, 150 is Bold, and 200 is ExtraBold — values that correspond to the CSS font-weight scale multiplied by roughly 1.1. These numeric values appear often enough in fontconfig output that they are worth memorizing.

To restrict output to specific properties, list them after the pattern:

```
# Show family name and file path for all fonts
fc-list : family file

# Show family, style, and language support
fc-list : family style lang
```

One practically useful query: checking whether a specific font family is installed before writing a document that depends on it.

```
fc-list | grep -i "EB Garamond"
```

If this produces no output, the font is not installed. If it does, you know the exact family name as fontconfig recognizes it — useful because the name in fontconfig may not exactly match the name on the font’s download page. Case differences, spacing, and hyphenation all matter when you specify a font in a LaTeX fontspec command or a CSS font-family declaration.

fc-match: resolving font requests

`fc-match` shows you which font file `fontconfig` would select for a given font request. This is the single most useful diagnostic tool in the `fontconfig` suite, because it reveals what will actually happen when a document requests a font — which may not be what you intended.

```
fc-match "EB Garamond"
```

```
EBGaramond-Regular.otf: "EB Garamond" "Regular"
```

If the requested font is not installed, `fontconfig` will substitute the closest match it can find:

```
fc-match "Georgia"
```

```
DejaVuSerif.ttf: "DejaVu Serif" "Book"
```

Georgia is not installed on this system (it is a Microsoft font, not commonly packaged for Linux). `Fontconfig` falls back to `DejaVu Serif`, which it considers the closest available serif face. This substitution happens silently in most applications — your document requests Georgia and gets `DejaVu Serif` without any warning. `fc-match` makes the substitution visible.

The `--format` flag controls what information is printed, using `fontconfig` property names enclosed in `%{}`:

```
fc-match --format="Family: %{family}\nFile:  %{file}\nWeight:  
→  %{weight}\nSlant:  %{slant}\n" "serif"
```

```
Family: DejaVu Serif  
File:   /usr/share/fonts/truetype/dejavu/DejaVuSerif.ttf  
Weight: 80  
Slant:  0
```

The generic family names `serif`, `sans-serif`, and `monospace` are resolved by `fontconfig` according to its substitution configuration. The system's default serif, sans-serif, and monospace fonts are typically configured in `/etc/fonts/conf.d/` — you can inspect those files to understand what your defaults are, or simply use `fc-match` to check them directly.

`fc-match` with the `-a` flag lists all matching fonts in preference order, not just the best match:

```
fc-match -a "serif" | head -10
```

This shows the full fallback chain — useful when building documents that must render correctly across a wide character range, because it reveals which fonts will be used for characters not covered by the primary face.

For diagnosing font problems in LaTeX or Pandoc pipelines, a reliable workflow is:

1. Check whether the font is known: `fc-list | grep -i "font name"`
2. Check what `fontconfig` resolves it to: `fc-match "font name"`
3. If the wrong font is being substituted, install the correct one and rebuild the cache

This three-step process resolves the majority of “I specified font X but my document used font Y” problems.

fc-query and fc-scan: inspecting font files

Where `fc-list` and `fc-match` work with the `fontconfig` catalog, `fc-query` and `fc-scan` work directly with font files and report their properties.

`fc-query` reads a font file and prints all the properties `fontconfig` can extract from it:

```
fc-query /usr/share/fonts/truetype/dejavu/DejaVuSans.ttf
```

The output is verbose — dozens of lines per font face, including character coverage, supported languages, all registered name strings, and metric values. For targeted inspection, use `--format` to extract specific properties:

```
fc-query --format="%{family}\n File:    %{file}\n Version:
↳  %{fontversion}\n Spacing:  %{spacing}\n Lang:    %{lang}\n" \
/path/to/font.ttf
```

The `lang` property is especially useful: it lists the language tags (BCP 47 codes) for all languages the font supports. A font claiming to support Arabic (`ar`) must include the Arabic Unicode block; `fc-query` lets you verify this before depending on it in a multilingual document.

`fc-scan` is similar to `fc-query` but also accepts directories, scanning all font files within them:

```
# Inspect all fonts in a directory
fc-scan ~/.local/share/fonts/

# With formatting
fc-scan --format="%{family}: %{file}\n" ~/.local/share/fonts/
```

This is useful when you have downloaded a font family with multiple files — regular, italic, bold, bold italic, and possibly condensed, extended, and variable variants — and want to confirm that all expected faces are present before building a document that uses them.

A practical use of `fc-query` beyond simple inspection: checking whether a font file contains genuine small capitals or whether an application would be forced to fake them. A font with real small capitals will have the `smcp` feature listed in its OpenType feature set. While `fontconfig` does not directly report OpenType features, you can use the `otfinfo` tool from the `lcdf-typetools` package:

```
otfinfo -f /path/to/font.otf
```

This lists all OpenType features the font supports — `liga`, `kern`, `onum`, `smcp`, and so on. Combine this with `fc-query` output to get a complete picture of a font’s capabilities before committing to it for a document.

Installing fonts manually

The most common source of fonts for serious typographic work is direct download from type foundries or font distribution sites. Google Fonts, Font Squirrel, the League of Moveable Type, and commercial foundries like Klim, Commercial Type, and Hoefler&Co all distribute fonts as downloaded archives. The installation process is the same regardless of source.

Step 1: Identify the target directory. For a single user, install to `~/.local/share/fonts/`. For system-wide installation (available to all users), install to `/usr/local/share/fonts/` (requires root). Create the directory if it does not exist:

```
mkdir -p ~/.local/share/fonts/
```

Step 2: Organize the files. Font families can contain many files. A convention that scales well is to create a subdirectory per family:

```
mkdir -p ~/.local/share/fonts/EBGaramond/  
cp ~/Downloads/EBGaramond/*.otf ~/.local/share/fonts/EBGaramond/
```

Keeping families in their own directories makes future management easier: removing a font means removing its directory, and inspecting what you have installed is a simple `ls`.

Step 3: Rebuild the cache:

```
fc-cache -f ~/.local/share/fonts/
```

Step 4: Verify the installation:

```
fc-list | grep -i garamond
```

If the font appears in the output, it is ready to use. If it does not, confirm the file path was correct and that `fc-cache` ran without errors.

For fonts distributed as packages by your system's package manager, the process is simpler: the package manager handles installation, the font lands in `/usr/share/fonts/`, and `fontconfig`'s cache is updated automatically as part of the package post-install scripts. On Debian and Ubuntu systems, many fonts are available as packages with the naming convention `fonts-*`:

```
# Search for available font packages
apt search fonts-

# Install EB Garamond
sudo apt install fonts-ebgaramond

# Install the Noto font family (broad Unicode coverage)
sudo apt install fonts-noto
```

The package manager route is preferable for fonts that are available as packages: it means the font is tracked by the package manager, will be updated when a new version is released, and will be present on any system that uses the same package list (useful for reproducible document builds).

For fonts that are not packaged — commercial fonts, recently released fonts, or fonts you have purchased from a foundry — manual installation to `~/.local/share/fonts/` is the right approach.

Previewing fonts without a GUI

Evaluating a typeface for a specific purpose requires seeing it set at the relevant sizes with representative text. Without a graphical font preview application, there are several CLI approaches.

Using **ImageMagick's `convert`** is the most portable approach: it renders text in a specified font to an image file that you can then view with any image viewer or include in a test document.

```
convert \  
-size 800x200 \  
-background white \  
-fill black \  
-font "EB-Garamond" \  
-pointsize 32 \  
label:"The quick brown fox jumps over the lazy dog." \  
preview-garamond.png
```

To preview multiple fonts for comparison, loop over them:

```
for font in "EB-Garamond" "Palatino" "DejaVu-Serif"; do  
  convert \  
  -size 800x60 \  
  -background white \  
  -fill black \  
  -font "$font" \  
  -pointsize 24 \  
  label:"$font: Pack my box with five dozen liquor jugs." \  
  "preview-{$font}.png"  
done
```

Note that ImageMagick uses `fontconfig` to resolve font names, so the name you pass to `-font` must match what `fontconfig` knows. Use `convert -list font` or `fc-match` to find the exact name.

Generating a specimen PDF with LaTeX gives you the most typographically accurate preview, because it uses the same rendering pipeline you will use in production:

```

\documentclass{article}
\usepackage{fontspec}
\setmainfont{EB Garamond}
\begin{document}

{\huge The CLI Typographer}

{\Large Pack my box with five dozen liquor jugs.}

{\normalsize The quick brown fox jumps over the lazy dog.
Sphinx of black quartz, judge my vow. How vexingly quick
daft zebras jump.}

{\small 0123456789 \quad fi fl ff ffi ffl}

{\footnotesize ABCDEFGHIJKLMNOPQRSTUVWXYZ\
abcdefghijklmnopqrstuvwxyz}

\end{document}

```

Compile with XeLaTeX (required for fontspec) and open the resulting PDF. This approach shows you exactly how the font will look in a document — including how the engine handles ligatures, how the typeface renders at body and display sizes, and whether any character is missing and being substituted.

Writing a shell script to generate comparison sheets scales this approach to evaluating multiple candidates at once:

```

#!/bin/sh
# font-compare.sh: generate a comparison PDF for a list of fonts

cat > /tmp/font-compare.tex << 'EOF'
\documentclass{article}
\usepackage{fontspec}
\usepackage{geometry}
\geometry{margin=20mm}
\newcommand{\showfont}[1]{%
  \begin{group}
    \setmainfont{#1}%
    \noindent{\large\textbf{#1}}\ \ [2pt]
    {\normalsize The quick brown fox jumps over the lazy dog.
fi fl ff ffi ffl - 0123456789}\ \ [8pt]
  \end{group}
}
EOF

```

```

\endgroup
}
\begin{document}
EOF

for font in "EB Garamond" "Palatino Linotype" "Libertinus Serif" \
"Cormorant Garamond" "Crimson Text"; do
  echo "\\showfont{${font}}" >> /tmp/font-compare.tex
done

echo '\end{document}' >> /tmp/font-compare.tex

xelatex -output-directory=/tmp /tmp/font-compare.tex

```

This produces a single PDF with one specimen per font, formatted consistently for direct comparison.

The **pango-view** tool, where available, renders text using the Pango library (which underpins GTK and GNOME applications) and can output to a PNG file:

```

pango-view \
  --font="EB Garamond 24" \
  --text="The quick brown fox jumps over the lazy dog." \
  --output=preview.png

```

Pango-view's rendering will differ slightly from LaTeX's, but it is faster and does not require a full LaTeX installation. It is useful for quick visual checks when you are not yet ready to write a document.

Font licensing basics

A font is software, and like all software it is subject to copyright and licensing terms. Using a font in a document has licensing implications that are easy to overlook and sometimes consequential.

The licenses that matter most for CLI typographers fall into a small number of categories.

The SIL Open Font License (OFL) is the most common free and open-source font license. It permits free use, modification, and redistribution of fonts, including embedding in documents, with two restrictions: you may not sell the fonts themselves as standalone products, and modified versions must be released under the OFL and must not use the original font’s reserved name. Most of the high-quality free fonts you will encounter — EB Garamond, Libertinus, Crimson, Lato, Raleway, the entire Google Fonts catalog with few exceptions — are released under the OFL. For typesetting documents you intend to distribute, fonts under the OFL are the simplest licensing situation: use them freely, embed them in PDFs freely, no further permission needed.

The GNU General Public License (GPL) and its font-specific variant, the **GPL with font exception**, are used by some open-source fonts including the GNU FreeFont family and some fonts distributed with TeX Live. The font exception (sometimes called the “GPL font exception”) clarifies that embedding a GPL font in a document does not cause the document itself to be covered by the GPL — without this exception, the GPL’s terms would technically require any document that embeds the font to be open-source itself. When a font uses the GPL without the font exception, embedding it in a commercial or proprietary document is legally ambiguous. In practice, most GPL-licensed fonts include the exception.

Commercial font licenses vary substantially between foundries. The typical commercial font license grants you the right to install the font on a specified number of computers and to use it in documents — but “use in documents” has important sub-clauses. Most commercial licenses permit *static embedding* in PDFs: the font can be embedded in a PDF for viewing and printing, which is what `pdflatex` and `xelatex` do by default. Fewer licenses permit *web embedding* via `@font-face` without a separate webfont license. Some licenses restrict commercial use. Some restrict modification. The specific terms depend on the foundry and the license tier you purchased.

For CLI typesetting specifically, the key questions to ask of any commercial font are:

1. **Desktop use:** May I install this font on my workstation and type-setting server?
2. **PDF embedding:** May I embed this font in PDFs for distribution? (Almost all commercial licenses say yes.)
3. **Server use:** If I am building documents in an automated pipeline on a server, is that covered by a desktop license? (Often no — many licenses restrict use to a named number of *desktops*, and a build server may require a separate server license.)
4. **Webfont use:** If I am generating HTML output with `@font-face`, is that covered? (Usually no — requires a separate webfont license.)

The server use question is the one that most often catches CLI typographers off guard. If you are building a PDF generation service — accepting user content and rendering it to PDF with a specific typeface — and that service runs on a server, a single-user desktop font license is unlikely to cover it. Fonts licensed under the OFL do not have this restriction.

System fonts — the fonts that come pre-installed with an operating system — carry the operating system’s licensing terms. On macOS, fonts like Helvetica Neue and San Francisco are licensed for use on Apple devices and in documents, but may not be redistributed as font files or used on non-Apple servers. On Windows, fonts like Times New Roman, Arial, and Georgia are licensed for use on Windows and in documents created on Windows, but their use in automated Linux-based pipelines is a gray area. The safest path for cross-platform document pipelines is to rely on OFL-licensed fonts rather than system fonts.

Font licensing and PDF embedding interact in a specific technical way. PDF files can embed fonts in two ways: full embedding (the entire font program is included in the PDF) or subsetting (only the glyphs used in the document are included). Subsetting produces smaller PDF files. Both embedding modes are permitted by most font licenses, but a few older commercial licenses prohibit full embedding while permitting subsetting. LaTeX’s PDF output tools embed fonts by default and can be configured to subset or fully embed. Checking that your fonts are correctly embedded is part of preparing a print-ready PDF, and we will cover this in Chapter 8.

The practical recommendation is straightforward: for serious typographic work, build a collection of OFL-licensed typefaces that covers your common document types — a text serif, a text sans-serif, a monospace, and perhaps a display face — and use these as your defaults. Supplement with commercial fonts where design requirements demand it, and read the license carefully before deploying to an automated pipeline.

A starting collection of high-quality OFL fonts that serve most CLI typesetting purposes well:

- **EB Garamond** — A revival of Claude Garamond’s 16th-century designs. Elegant, scholarly, excellent for long-form text and academic documents.
- **Libertinus Serif** — A fork of Linux Libertine with improved metrics and broader OpenType support. A reliable, neutral text face.
- **Source Serif (Adobe)** — Designed specifically for screen readability at text sizes. Clean and unpretentious.
- **IBM Plex Serif, Sans, and Mono** — A complete type family released under the OFL by IBM. Consistent across all three styles, professional, and well-hinted for screen use.
- **Fira Sans and Fira Mono** — Designed for Firefox OS, now a mature and capable sans-serif family with a matching monospace.
- **Noto (Google)** — Designed to support all Unicode scripts. If your document includes non-Latin text, Noto covers it. Available in serif, sans-serif, and monospace variants for dozens of scripts.
- **JetBrains Mono** — A monospace face specifically designed for code display, with features like ligatures for common programming symbols and careful optical sizing for long reading sessions.

All of these are available through package managers on most Linux distributions and as direct downloads from their respective foundries and Google Fonts. They form a foundation on which the document examples in Part IV will build.

With fonts installed, queryable, and understood, we have the complete foundation for practical work. Part I has covered where typography comes from, what it consists of, how digital type works, and how to manage it on a real system. Part II puts these tools into motion.

Markdown as Source: The Modern Workflow

Markdown for Document Authors

Markdown is, at first glance, a formatting shorthand for the web — a way to write **`**bold**`** and have it render as **bold**, to write `# Heading` and have it become an `<h1>`. That characterisation is accurate but limiting. In the hands of the CLI typographer, Markdown is something more: a source format for serious documents, one that can be compiled to PDF, HTML, EPUB, and DOCX from a single file, that integrates with bibliography managers, supports footnotes and cross-references and tables, and scales to book-length projects.

The key to this larger role is Pandoc, which we will cover in depth in the next chapter. But to use Pandoc well — to write Markdown that produces the output you intend — you first need to understand what Markdown is, which of its several dialects you are working in, and what it can and cannot express. That is what this chapter covers.

The Markdown landscape: CommonMark, GFM, and Pandoc Markdown

Markdown was created by John Gruber in 2004 as a text-to-HTML conversion tool. The original specification, published on Gruber’s website, was informal and in some respects ambiguous. Over the following decade, as Markdown was adopted by platforms ranging from GitHub to Reddit to Stack Overflow, dozens of implementations emerged — each resolving the ambiguities differently, each adding its own extensions. By the early 2010s, “Markdown” was less a single format than a family of related-but-incompatible dialects.

CommonMark, published in 2014 by a group of developers including Jeff Atwood and John MacFarlane, was an attempt to resolve this fragmentation with a rigorous formal specification. CommonMark defines precise rules for every edge case in the original Markdown syntax — how nested lists work, what happens when you mix blockquotes and code blocks, how link definitions interact with emphasis markers — tested against a comprehensive suite of examples. It is the closest thing Markdown has to a standard, and it forms the base of most modern implementations.

GitHub Flavored Markdown (GFM) is a CommonMark superset used by GitHub, GitLab, and many other platforms. It adds several extensions to CommonMark: tables (using the pipe syntax you likely already know), task lists (checkboxes in list items), strikethrough, and autolinks. GFM is what most developers encounter first and what most people mean colloquially when they say “Markdown.”

Pandoc Markdown is the dialect used by Pandoc, and it is the one this book uses throughout. It is a CommonMark superset with an extensive set of optional extensions that add nearly every feature a typographer might need: footnotes, definition lists, metadata blocks, citations, mathematical notation, cross-references, fenced divs for custom styling, line blocks, and more. The full list of Pandoc Markdown extensions runs to over sixty items; the defaults cover the needs of most documents.

When you run Pandoc with `-f markdown` (or simply provide a `.md` file), you get Pandoc Markdown with its default extension set enabled. You can add or remove specific extensions by appending them to the format name with `+` or `-`:

```
# Enable smart quotes and disable pipe tables
pandoc -f markdown+smart-pipe_tables input.md -o output.pdf
```

For most purposes, the default extension set is appropriate and you do not need to adjust it. Where specific extensions are relevant to the features described in this chapter, they are noted.

A practical note on source file encoding: always write Markdown source files in UTF-8. This is the default in every modern text editor and is

assumed by all the tools in this book. If you work with a legacy editor or encounter files from older systems, verify the encoding with `file your-document.md` — it should report “UTF-8 Unicode text.”

Front matter and metadata with YAML

A Pandoc Markdown document begins with a *metadata block* — a YAML section delimited by lines containing only three dashes (---). This block contains document-level information: the title, author, date, language, bibliography, and a wide range of settings that control how the document is rendered.

```
---
title: "On the Typographic Quality of CLI-Produced Documents"
subtitle: "A Survey of Current Practice"
author:
  - "A. N. Author"
  - "B. M. Collaborator"
date: 2024-03-15
lang: en-GB
---
```

The YAML metadata block must appear at the very beginning of the file — before any other content. If it appears later, Pandoc treats it as a fenced code block rather than metadata. The closing --- (or . . .) ends the block and the document body begins immediately after.

YAML has its own syntax rules that interact with Pandoc in ways worth understanding.

Strings can be unquoted if they contain no special characters, single-quoted if they contain only single-escape characters, or double-quoted if they contain backslashes or other special sequences. Titles containing colons or other punctuation must be quoted:

```
title: "Typography: A Practical Guide" # correct
title: Typography: A Practical Guide # YAML parse error
```

Lists can be written in flow style (inline) or block style (one item per line):

```
# Flow style
keywords: [typography, CLI, pandoc, LaTeX]

# Block style
keywords:
  - typography
  - CLI
  - pandoc
  - LaTeX
```

Both are equivalent. Block style is more readable for longer lists.

Multi-line strings use YAML block scalars. The `|` operator preserves line breaks (useful for abstracts and descriptions); `>` folds line breaks into spaces (useful for long single-paragraph values):

```
abstract: |
  This paper examines the typographic quality of documents produced
  using CLI-based typesetting tools. We find that with appropriate
  configuration, CLI-produced documents match or exceed the quality
  of those produced by graphical applications.

description: >
  A very long description that continues across
  multiple source lines but renders as a single
  paragraph with normal word wrapping.
```

Structured author information is useful when the output format supports it — for instance, when generating academic papers with author affiliations, or conference proceedings with ORCID identifiers:

```
author:
  - name: "A. N. Author"
    affiliation: "University of Example"
    email: "author@example.edu"
    orcid: "0000-0000-0000-0001"
  - name: "B. M. Collaborator"
    affiliation: "Institute of Advanced Typography"
```

Whether these sub-fields are used depends on the Pandoc or Quarto template for your output format. Backend-specific templates may use name and affiliation directly; custom templates can use any fields you define.

PDF backend and font settings are set in the metadata block, allowing you to control the PDF layer without scattering those decisions through the manuscript source:

```
pdf-engine: typst
geometry: "margin=25mm, bindingoffset=10mm"
mainfont: "EB Garamond"
sansfont: "Fira Sans"
monofont: "JetBrains Mono"
fontsize: 12pt
linestretch: 1.25
```

The `mainfont`, `sansfont`, and `monofont` fields are used directly by modern PDF backends such as Typst and by Unicode-capable LaTeX engines via `fontspec`. If you are using pdfLaTeX for compatibility reasons, use `fontfamily` and the appropriate LaTeX package name instead.

Table of contents generation is controlled by the `toc` flag:

```
toc: true
toc-depth: 2
numbersections: true
```

Bibliography settings connect the document to an external bibliography file and citation style:

```
bibliography: references.bib
csl: chicago-author-date.csl
link-citations: true
```

We will cover bibliographies in detail later in this chapter and in Chapter 22 (Books).

Variable files and defaults are a Pandoc feature that lets you move shared metadata out of individual documents and into a reusable defaults file. If you are producing a series of documents with the same settings — the same fonts, the same geometry, the same bibliography — create a file called `defaults.yaml`:

```
# defaults.yaml
from: markdown
to: pdf
pdf-engine: xelatex
metadata:
  documentclass: article
  mainfont: "EB Garamond"
  geometry: "margin=25mm"
  bibliography: ~/documents/references.bib
```

Then invoke Pandoc with `--defaults defaults.yaml` to apply these settings to any document. Per-document metadata blocks override defaults when both specify the same field.

Basic Markdown syntax

Before the extensions, the core. Pandoc Markdown inherits all of CommonMark's syntax, which covers the constructions most writers need daily.

Headings use the ATX style — hash symbols followed by the heading text — or the setext style (underlines of = or - for levels 1 and 2). ATX is preferred because it generalises to all heading levels:

```
# Part title (level 1)
## Chapter title (level 2)
### Section (level 3)
#### Subsection (level 4)
```

Emphasis and strong emphasis use asterisks or underscores. Pandoc treats single delimiters as italic and double as bold:

```
This is italic and this is bold.
```

The choice between asterisks and underscores is largely stylistic, but be consistent within a project. Pandoc Markdown by default requires word-boundary conditions to be met for underscore-delimited emphasis to activate — `intraword_underscores` is enabled — which means `file_name_here` is not inadvertently italicised.

Block quotations use `>` at the start of each line:

```
> Typography is the craft of endowing human language with a durable
> visual form, and thus with an independent existence.
>
> - Robert Bringhurst
```

Code uses backtick delimiters for inline code and triple backticks (or four-space indentation) for fenced code blocks. Fenced blocks accept a language identifier for syntax highlighting:

The command ``fc-list`` lists installed fonts.

```
```sh
fc-list | grep -i garamond
```
```

Lists are formed with `-`, `*`, or `+` for unordered lists, and numerals followed by `.` or `)` for ordered lists. Pandoc supports *fancy lists* — ordered lists that can start at any number and use letters or Roman numerals — via the `fancy_lists` extension, which is enabled by default:

```
- Unordered item
- Another item
  - Nested item (two-space or four-space indent)

1. First item
2. Second item
  a. Sub-item using letter numbering
  b. Another sub-item
```

```
(i) Roman numeral list
(ii) Second item
```

Horizontal rules are three or more hyphens, asterisks, or underscores on a line by themselves. In LaTeX output, a horizontal rule becomes an `\hrule` or `\rule`; in HTML output, it becomes `<hr>`. Use them sparingly — as visual breaks between major sections of content, not as decoration.

Links and images use the same square-bracket/parenthesis syntax:

```
See the [Pandoc documentation](https://pandoc.org/MANUAL.html).
```

```
![Alt text for the figure](path/to/image.png){width=80%}
```

The attribute syntax `{width=80%}` is a Pandoc extension that passes attributes to the image element in HTML or to a `\includegraphics` call in LaTeX. In LaTeX output, images are automatically wrapped in a `figure` environment with a caption derived from the alt text.

Footnotes

Footnotes are the workhorse of scholarly and technical prose — the mechanism for providing supplementary information, source citations in note-based styles, and parenthetical asides that would interrupt the flow of the main text. Pandoc Markdown supports two footnote styles.

Reference footnotes separate the marker from the content:

```
Typography has been described as the craft of endowing human language
with a durable visual form.[^bringhurst]
```

```
[^bringhurst]: Bringhurst, Robert. *The Elements of Typographic Style*,
  4th ed. Hartley & Marks, 2012. The full definition continues: "and
↪ thus
  with an independent existence."
```

The marker can be any string — a number, a word, an abbreviation. Pandoc renumbers footnotes automatically in the output, so you do not need to maintain sequential numbers in the source. Reference footnote definitions can appear anywhere in the document — at the bottom of the relevant section, at the end of the chapter, or collected at the end of the file — and Pandoc will place them correctly in the output.

Inline footnotes embed the content directly at the point of use, which is convenient for short asides:

```
The em dash – used for parenthetical remarks^[The en dash is used for ranges, such as pages 12--34; the em dash for interruptions or asides.] – is one of the most misused marks in typography.
```

Both styles produce identical output; the choice is a matter of authorial preference. For long footnotes with multiple paragraphs or nested markup, reference style is cleaner in the source; for short factual notes, inline style keeps the annotation close to what it annotates.

In LaTeX output, footnotes become `\footnote{}` commands and are rendered at the bottom of the page where they occur. In HTML output, they become numbered superscript markers linking to a list of notes at the end of the document section. In EPUB output, they become pop-up notes where the format supports it.

Citations

Pandoc's citation system is one of its most powerful features, and one of the primary reasons to use Pandoc Markdown for academic and technical writing rather than writing directly in LaTeX.

The system works through three components: a bibliography file (typically BibTeX `.bib` format, though many other formats are supported), a citation style file (CSL — Citation Style Language — a widely-used XML format with thousands of available styles), and inline citation markers in the document.

A minimal bibliography file `references.bib` looks like this:

```
@book{bringhurst2004,
  author = {Bringhurst, Robert},
  title  = {The Elements of Typographic Style},
  edition = {4},
  publisher = {Hartley & Marks},
  year    = {2012}
}

@book{knuth1984,
  author = {Knuth, Donald E.},
  title  = {The TeXbook},
  publisher = {Addison-Wesley},
  year    = {1984}
}
```

Citation markers in the document use the @key syntax, where the key matches the first field of the BibTeX entry:

```
The typographic scale has been extensively documented [@bringhurst2004,
↪ pp. 29--32].
```

```
As Knuth himself explains [-@knuth1984, ch. 14], the line-breaking
algorithm evaluates entire paragraphs, not individual lines.
```

```
@bringhurst2004 [p. 17] opens his definition of typography with the
observation that text has both functional and aesthetic dimensions.
```

The three forms do different things:

- `[@bringhurst2004]` — parenthetical citation: (Bringhurst 2012) in author-date styles
- `[-@knuth1984]` — suppresses the author name, producing only (1984) — useful when the author has just been named in the sentence
- `@bringhurst2004` — inline citation: Bringhurst (2012) in author-date styles, used when the author is grammatically part of the sentence

Multiple citations in a single pair of brackets are separated by semicolons:

```
Several authors have addressed this question [@bringhurst2004; @knuth1984;
↔ @lampo1994].
```

To process citations, Pandoc must be invoked with the `--citeproc` flag:

```
pandoc --citeproc --bibliography references.bib --csl
↔ chicago-author-date.csl \
input.md -o output.pdf
```

Or equivalently, with the relevant fields in the metadata block:

```
---
bibliography: references.bib
csl: chicago-author-date.csl
---
```

CSL files for thousands of citation styles — Chicago, APA, MLA, Vancouver, Harvard, IEEE, Nature, and journal-specific styles — are available from the Zotero Style Repository at zotero.org/styles. Download the appropriate `.csl` file, place it alongside your document or in a shared directory, and reference it in the metadata block.

The bibliography is automatically appended to the document. To control where it appears — at the end of a specific section rather than the very end of the document — use a `div` with the `refs` class:

```
## References

::: {#refs}
:::

## Appendix

Content after the bibliography.
```

Tables

Pandoc Markdown supports four table syntaxes of varying complexity. Understanding when to use each saves considerable formatting effort.

Pipe tables are the most readable in source form and appropriate for most cases:

| Typeface | Classification | Best use |
|----------------|----------------|--------------------|
| EB Garamond | Humanist serif | Body text, books |
| Fira Sans | Humanist sans | Screen, UI text |
| JetBrains Mono | Monospace | Code, terminals |
| Bodoni | Didone/modern | Display, headlines |

: Recommended typefaces by use case {#tbl:typefaces}

The alignment of column separators in the header row controls text alignment: :--- for left, :---: for centred, ---: for right. Columns without explicit alignment default to left. The caption, if present, follows the table preceded by : and a space. The {#tbl:label} attribute assigns a cross-reference identifier.

Simple tables forgo the pipe syntax and use spacing to imply column structure — readable in source but fragile if column widths vary:

| Right | Left | Center | Default |
|-------|------|--------|---------|
| 12 | 12 | 12 | 12 |
| 123 | 123 | 123 | 123 |
| 1 | 1 | 1 | 1 |

: Demonstration of simple table syntax.

Multiline tables use a header of dashes above and below the column headers and allow cell content to wrap across multiple lines — useful for tables with verbose cell content:

```

-----
| Fruit   | Color   | Notes |
-----
Apple	Red, green	Harvested in autumn;
		many varieties available
Banana	Yellow	Best at peak ripeness;
		green bananas are starchy
-----

: A multiline table with wrapped cells

```

Grid tables offer full control over alignment and cell content at the cost of verbose source syntax:

```

+-----+-----+-----+
| Fruit   | Price   | Advantages |
+-----+-----+-----+
| Bananas | $1.34   | - built-in wrapper |
|         |         | - bright colour   |
+-----+-----+-----+
| Oranges | $2.10   | - cures scurvy |
|         |         | - tasty         |
+-----+-----+-----+

```

Grid tables support arbitrary block content in cells — lists, code blocks, even nested tables — and are the right choice when cell content is complex. The tradeoff is that they are tedious to maintain by hand. In practice, most tables in technical documents fit comfortably in pipe table syntax, and grid tables are reserved for the occasional case where a cell needs multi-paragraph or list content.

In LaTeX output, all four table syntaxes produce `longtable` environments by default, which are capable of spanning multiple pages. Column alignment and the table caption are preserved. In HTML output, they produce standard `<table>` elements.

Cross-references

Cross-references — “see Figure 3” or “as shown in Table 2.1” — are straightforward in LaTeX (using `\label` and `\ref`) but require a Pandoc extension called `pandoc-crossref`, a filter that adds cross-reference syntax to Pandoc Markdown.

Install `pandoc-crossref` from your package manager or from its GitHub releases page, then add it to your Pandoc invocation:

```
pandoc --filter pandoc-crossref input.md -o output.pdf
```

With `pandoc-crossref`, labels are assigned using attribute syntax on headings, figures, tables, and equations:

```
## The typographic scale {#sec:scale}

![[Comparison of typefaces at equal point
↪ size](typefaces.png){#fig:typefaces}

Metric	Value
Em	12pt
En	6pt

: Standard typographic units {#tbl:units}

$$ E = mc^2 $$ {#eq:einstein}
```

References use a `@` prefix with the label:

```
As discussed in @sec:scale, the choice of type scale affects hierarchy.
The comparison in @fig:typefaces illustrates this clearly.
See @tbl:units for the standard metric values.
```

`pandoc-crossref` handles numbering automatically and produces correct output for both PDF (using LaTeX `\ref` and `\label`) and HTML (using anchor links). It also supports list-of-figures and list-of-tables generation, controlled through metadata variables.

For documents not using `pandoc-crossref`, basic heading cross-references can be achieved with Pandoc's built-in implicit header identifiers: every heading gets an anchor based on its text, accessible via standard Markdown link syntax:

```
## The typographic scale

...later in the document...

As discussed in [the typographic scale section](#the-typographic-scale),
hierarchy emerges from contrast.
```

This approach works well for HTML output and simple documents. For complex documents with figures, tables, and equations, `pandoc-crossref` is worth the setup cost.

Definition lists and other block elements

Pandoc Markdown includes several block-level constructs beyond the CommonMark standard.

Definition lists associate terms with their definitions — useful for glossaries, term explanations, and structured reference content:

```
Kerning
: The adjustment of space between specific letter pairs to correct
  for optical irregularities caused by their shapes.

Leading
: The vertical distance between the baselines of successive lines
  of type. Named for the lead strips placed between lines in
  hot-metal composition.

Tracking
: Uniform adjustment of inter-character spacing across a range of
  text. Distinguished from kerning by its global rather than
  pair-specific application.
```

Each term is followed by one or more definitions, each beginning with `:` and at least one space. Multiple definitions for a single term are permitted.

Line blocks preserve line breaks and leading spaces, making them useful for poetry, addresses, and formatted plain text:

```
| 742 Evergreen Terrace
| Springfield, USA 12345
|
| 15 March 2024
```

The leading `|` and space signals a line block; each line is preserved exactly.

Fenced divs, introduced with the `fenced_divs` extension (enabled by default in Pandoc Markdown), create block-level elements with custom classes and attributes. These pass through to HTML as `<div>` elements and, with appropriate LaTeX template support, can be rendered as custom environments:

```
::: {.callout type="note"}
**Note:** The `fontspec` package requires XeLaTeX or LuaLaTeX.
Using it with pdfLaTeX will produce an error.
:::

::: {.aside}
Sidebar content that appears in the margin or a callout box
depending on the output format and template.
:::
```

The power of fenced divs emerges in combination with custom Pandoc templates and LaTeX environments: you define an environment in your LaTeX preamble, give it a class name, and Pandoc maps the fenced div to that environment automatically. We will use this technique extensively in Chapter 24 when building templates and style systems.

Structuring long documents

A single Markdown file works well for documents up to perhaps ten or twenty thousand words. Beyond that, practical considerations — navigation, collaboration, build times, version control diff legibility — argue for splitting the document into multiple files, one per chapter or major section.

Pandoc handles multi-file documents in the simplest possible way: you pass multiple input files on the command line, and Pandoc concatenates them in order before processing:

```
pandoc metadata.yaml \  
  chapters/01-introduction.md \  
  chapters/02-history.md \  
  chapters/03-typography-fundamentals.md \  
  chapters/04-digital-concepts.md \  
  -o book.pdf
```

The first argument, `metadata.yaml`, is a standalone YAML file containing the document's front matter — the same YAML that would appear in a metadata block, but extracted to its own file so that it does not need to be repeated or copied between chapter files. This is the standard approach for book-length projects.

Each chapter file contains only its content, beginning with its chapter-level heading:

```
# The Origins of Typography  
  
The story of typesetting begins...
```

There is no need for YAML metadata in individual chapter files, though you can include chapter-specific metadata (epigraphs, chapter-level abstracts, or custom settings) if the template supports it.

A typical project directory for a book might look like this:

```

book/
  metadata.yaml      # Document-wide settings and metadata
  references.bib     # Bibliography
  styles/
    book.typ        # Custom Typst template
    book.latex      # LaTeX fallback template
    book.css        # CSS for HTML output
  figures/
    ch01-timeline.pdf
    ch03-type-anatomy.svg
  chapters/
    00-preface.md
    01-history.md
    02-fundamentals.md
    03-digital-concepts.md
    04-font-management.md
  Makefile          # Build rules (see Chapter 10)

```

The Makefile assembles the final document from these components, which we cover in Chapter 10. For now, note that the file structure is simple: flat source files with a clean naming convention, a shared metadata file, and a shared bibliography. There is nothing framework-specific about it — the same structure works regardless of whether the output is PDF, HTML, or EPUB.

Heading levels and document structure interact with the output format in ways worth planning for. In Pandoc’s default behaviour, # headings become `\chapter` commands in LaTeX book or memoir document classes, and `\section` commands in article. If you use `documentclass: book` in your metadata, your chapter-level Markdown heading should be #, not ##. If you use `documentclass: article`, your top-level heading is a section and # becomes `\section`.

This can be adjusted with the `--shift-heading-level-by` flag:

```

# Treat # as chapter, ## as section (book class)
pandoc --shift-heading-level-by=0 ...

```

```
# Treat # as section (article class, the default)
pandoc --shift-heading-level-by=-1 ...
```

For most projects, the simplest approach is to decide at the outset whether you are writing an article or a book, set `documentclass` accordingly, and use `#` for the top structural level throughout.

Parts, in a book with multiple parts each containing multiple chapters, can be expressed by placing a special `# Part Title {.unnumbered .part}` heading before the chapters belonging to that part. With appropriate template support, this becomes a `\part` command in LaTeX. Without template support, it appears as a regular heading. In Quarto, which we cover in Chapter 14, part structure is handled through the project configuration rather than through Markdown syntax.

Markdown's apparent simplicity is deceptive. What appears to be a minimal plain-text format is, in Pandoc's hands, a fully capable authoring language for documents of any length and complexity. The features covered in this chapter — metadata, footnotes, citations, tables, cross-references, and multi-file structure — are sufficient to write and publish the book you are reading now, or an academic paper, or a technical manual. The next chapter covers the engine that converts this source into polished output.

Pandoc — The Universal Converter

In 2006, John MacFarlane, a philosopher at UC Berkeley, released the first version of a command-line tool he had written for his own use. He needed to convert documents between formats — from Markdown to HTML, from HTML to LaTeX, between various academic document types — and found the existing tools inadequate. The tool he wrote, Pandoc, has since become the backbone of a substantial fraction of all CLI document production in the world.

The name means what it sounds like: *pan* (all) *doc* (document). Pandoc reads documents in dozens of input formats and writes them to dozens of output formats. At the time of writing, it understands over thirty input formats — Markdown, HTML, LaTeX, DOCX, EPUB, reStructuredText, Org Mode, Jupyter notebooks, and many more — and can write to over forty output formats. But the number of formats is not what makes Pandoc significant. What makes it significant is the model underneath: the abstract syntax tree.

This chapter covers how Pandoc works, how to use it well, and how to extend it. Chapter 7 and Chapter 8 go deeper on its two most important output formats, HTML and PDF.

How Pandoc thinks: the AST model

Most document conversion tools work by pattern matching: they look for strings that look like Markdown headings and emit strings that look like HTML headings, or LaTeX headings, or whatever the target format requires. This approach works reasonably well for simple documents and breaks down on anything complex, because it has no understanding of the document's structure — only its surface form.

Pandoc takes a different approach. When it reads a document, it parses it into an *abstract syntax tree* (AST) — an in-memory representation of the document's structure, independent of any particular format. Every element of the document is represented as a node in this tree: headers, paragraphs, emphasis, lists, blockquotes, code blocks, tables, footnotes, citations, images, and so on. The tree describes what the document *is*, not what it looks like in any particular format.

Once the document is in AST form, Pandoc can write it to any output format by *serialising* the AST according to that format's rules. A Header node at level 2 becomes `<h2>` in HTML, `\subsection{}` in LaTeX, `##` in Markdown, `^` in RST — whichever representation the output format uses for a second-level heading.

You can inspect the AST directly using Pandoc's native output format:

```
echo "# Hello\n\nA paragraph with *emphasis*." | pandoc -f markdown -t
↳ native
```

```
[ Header 1 ("hello",[],[]) [Str "Hello"]
, Para [Str "A",Space,Str "paragraph",Space,Str "with",Space
,Emph [Str "emphasis"],Str "."]
]
```

The tree is a list of block elements. The Header node carries its level (1), an identifier tuple (the auto-generated anchor "hello", plus empty lists for classes and key-value attributes), and a list of inline content. The Para node carries a list of inlines: individual words as Str nodes, spaces as Space nodes, and the italicised word as an Emph node wrapping a Str.

This representation is precise and complete. Every detail that Pandoc can represent is in the AST. And this is where Pandoc's extension mechanism enters the picture: because everything passes through the AST, you can intercept and transform the AST between parsing and serialisation. Those transformations are *filters*, and they are what makes Pandoc genuinely programmable.

The practical implication of the AST model for everyday use is this: Pandoc's conversions are lossless within the AST's capabilities. Information that the AST can represent is preserved across conversions. Information that the AST cannot represent — format-specific features with no equivalent in other formats — is either approximated or lost. A LaTeX `\marginpar{}` command, for instance, has no counterpart in HTML's document model, so Pandoc cannot convert it meaningfully to HTML. Understanding what the AST can and cannot represent helps you predict which conversions will work cleanly and which will require manual adjustment.

The basic invocation

Pandoc is invoked from the command line with at minimum an input file (or standard input) and either an output file or an explicit output format:

```
pandoc input.md -o output.pdf
pandoc input.md -o output.html
pandoc input.md -o output.docx
pandoc input.md -t latex
```

When `-o` specifies an output file, Pandoc infers the output format from the file extension. When it cannot infer the format, or when you want to be explicit, use `-t` (or `--to`):

```
pandoc input.md -t html -o output.html # explicit format
pandoc input.md -t html5 # HTML5 specifically
pandoc input.md -t latex > output.tex # to stdout
```

Similarly, `-f` (or `--from`) specifies the input format explicitly, which is necessary when reading from standard input or when the file extension is ambiguous:

```
cat input.md | pandoc -f markdown -t html
pandoc -f latex -t markdown input.tex -o input.md
```

Pandoc can read from multiple input files, concatenating them in order before processing. This is the mechanism for multi-file book projects as described in Chapter 5:

```
pandoc metadata.yaml ch01.md ch02.md ch03.md -o book.pdf
```

A YAML file passed as input is treated as a metadata source, not as body content. Its values are merged with any YAML front matter in the Markdown files, with command-line variables taking highest precedence.

Essential flags

Pandoc has over a hundred command-line options. The following are the ones you will use constantly.

--standalone / **-s** produces a complete, self-contained document rather than a fragment. Without this flag, HTML output is a bare `<body>` fragment with no `<html>`, `<head>`, or `<style>` elements. With it, you get a complete HTML file. For PDF and DOCX output, `--standalone` is implied and has no additional effect. For HTML output intended for embedding in a larger site, omit it; for a standalone HTML document, include it:

```
pandoc input.md -t html --standalone -o output.html
```

--toc / **--table-of-contents** inserts a table of contents, generated automatically from the document's headings. For HTML output, it becomes a `<nav>` element at the top of the body. For LaTeX/PDF output, it emits a `\tableofcontents` command:

```
pandoc input.md -o output.pdf --toc --toc-depth=2
```

`--toc-depth` controls how many heading levels are included; the default is 3.

`--number-sections` numbers headings automatically, adding 1, 1.1, 1.2.3 prefixes. In LaTeX output, this toggles `\setcounter{secnumdepth}`:

```
pandoc input.md -o output.pdf --number-sections
```

`--pdf-engine` selects the PDF renderer when the output format is PDF:

```
pandoc input.md -o output.pdf --pdf-engine=typst
pandoc input.md -o output.pdf --pdf-engine=xelatex
pandoc input.md -o output.pdf --pdf-engine=lualatex
pandoc input.md -o output.pdf --pdf-engine=pdfelatex
pandoc input.md -o output.pdf --pdf-engine=wkhtmltopdf
```

For PDF-only print work, `typst` is now the best default unless you are tied to a LaTeX class or package stack. If you are using `mainfont`, `sansfont`, or `monofont` with a LaTeX backend, you must use `xelatex` or `lualatex` — pdfLaTeX does not support `fontspec`. We discuss the tradeoffs in Chapter 8.

`--highlight-style` controls syntax highlighting in code blocks. Pandoc uses the KDE syntax highlighting library internally and ships with several built-in themes:

```
pandoc input.md -o output.html --highlight-style=tango
pandoc input.md -o output.html --highlight-style=breezedark
pandoc input.md -o output.pdf --highlight-style=monochrome
```

Available styles include `pygments` (the default), `tango`, `espresso`, `zenburn`, `kate`, `monochrome`, `breezedark`, and `haddock`. For print output, `monochrome` is usually the right choice — coloured code blocks render

poorly in greyscale. For screen output, choose whichever suits the document's visual design.

-V key=value (or `--variable key=value`) sets a template variable from the command line, overriding any value in the document's metadata block. This is useful for one-off adjustments without editing the source:

```
pandoc input.md -o output.pdf -V fontsize=11pt -V geometry=a5paper
```

--citeproc activates Pandoc's built-in citation processor, which processes @key citation markers against a bibliography file and formats them according to a CSL style. This must be combined with either `--bibliography` on the command line or a bibliography field in the metadata:

```
pandoc input.md -o output.pdf --citeproc \
  --bibliography=references.bib \
  --csl=chicago-author-date.csl
```

--include-in-header, **--include-before-body**, and **--include-after-body** inject the contents of a file into specific parts of the output document without requiring a full custom template:

```
# Add custom CSS to HTML output
pandoc input.md -t html --standalone \
  --include-in-header=custom.css \
  -o output.html

# Add a custom LaTeX preamble fragment
pandoc input.md -o output.pdf \
  --include-in-header=preamble-additions.tex
```

This is often the right approach for modest customisations. Custom templates (covered later in this chapter) are more powerful but more complex to maintain.

PDF engines and when to use each

When the output format is PDF, Pandoc converts the document to an intermediate format and then invokes an external tool to produce the final PDF. The choice of PDF engine is one of the most consequential decisions in a Pandoc workflow.

Typst is the best default for PDF-only print-oriented work when you are not constrained by legacy infrastructure. It gives you a modern PDF composition engine, strong typography, fast compilation, and a cleaner template story than LaTeX. If your goal is a print-ready PDF and not a publisher-mandated `.tex` file, start here.

pdfLaTeX is the oldest and most portable of the LaTeX PDF engines. It uses 8-bit input encoding (UTF-8 with the `inputenc` package), produces highly compatible PDF output, and compiles faster than the Unicode engines. Its primary limitation is font access: it can only use fonts that have been prepared in LaTeX's internal format, which in practice means the packaged fonts available through TeX Live (Palatino, Times, Helvetica, Charter, and many others via packages like `mathpazo`, `tgpage1a`, `helvet`, and `charter`). If your document's font requirements are covered by these packages, pdfLaTeX is a fine choice. If you need a specific OTF or TTF font — anything you would install via `fontconfig` — you need XeLaTeX or LuaLaTeX.

XeLaTeX uses Unicode input natively, supports OpenType and TrueType fonts directly via the `fontspec` package, and handles right-to-left scripts and complex script rendering through the HarfBuzz shaping engine. It is the standard choice for documents that use non-Latin scripts, require specific professional typefaces, or need access to advanced OpenType features. XeLaTeX compiles somewhat more slowly than pdfLaTeX and produces slightly larger PDF files.

LuaLaTeX shares XeLaTeX's Unicode support and font capabilities and adds the ability to embed Lua scripts within the LaTeX document itself. It is the engine used by the `microtype` package's most advanced features and by several sophisticated packages that require programmatic control over typesetting. LuaLaTeX compiles the most slowly of the three but

provides the most capability. For most documents, the choice between XeLaTeX and LuaLaTeX is not critical; choose XeLaTeX unless you have a specific reason to prefer LuaLaTeX.

wkhtmltopdf takes a fundamentally different approach: it renders the document as HTML using the WebKit browser engine and then converts the rendered page to PDF. The result looks like a browser rendering rather than a LaTeX rendering — which is sometimes exactly what is wanted. HTML/CSS-based document styles translate directly; web fonts work without configuration; CSS page media queries control the print layout. The tradeoff is that typography is web-grade rather than TeX-grade: no paragraph-level optimisation, no microtypography, no access to TeX's mathematical typesetting. For documents that are primarily styled with CSS and do not require TeX's mathematical capabilities, **wkhtmltopdf** is a practical choice.

WeasyPrint is a Python-based alternative to **wkhtmltopdf**, using a cleaner implementation of CSS Print and Paged Media standards. It is actively maintained, supports CSS features that **wkhtmltopdf** does not, and is the subject of Chapter 8's section on web-engine PDF generation.

The practical decision tree: start with Typst for PDF-only print work; switch to XeLaTeX or LuaLaTeX when you need a LaTeX-specific class, package, or submission format; use pdfLaTeX only for established compatibility workflows; reach for **wkhtmltopdf** or **WeasyPrint** when your document is designed in HTML/CSS and does not need a print-focused composition engine.

Templates

Pandoc's default output for any format is produced by a built-in template — a file written in Pandoc's template language that specifies the structure of the output and the positions where document content and metadata are inserted. You can inspect the default template for any output format:

```
pandoc --print-default-template=html > default.html
pandoc --print-default-template=latex > default.latex
```

Reading the default template for your target format is one of the most educational things you can do with Pandoc. It shows you exactly what variables are available (everything inside `$. . . $`), what conditionals the template uses, and where your content ends up. The default LaTeX template, for instance, shows the full sequence of packages loaded, the font configuration commands, the geometry settings, and the document structure — all driven by variables you can set in the YAML metadata block.

A Pandoc template is a text file with template syntax intermixed:

- `$variable$` — inserts the value of a metadata variable
- `$if(variable)$. . . $endif$` — conditional block
- `$for(list)$. . . $endfor$` — loop over a list
- `$body$` — inserts the converted document body

Here is a minimal HTML template that demonstrates all four constructs:

```
<!DOCTYPE html>
<html lang="$lang$" >
<head>
  <meta charset="utf-8">
  <title>$title</title>
  $if(css)$
  $for(css)$
    <link rel="stylesheet" href="$css">
  $endfor$
  $endif$
</head>
<body>
  $if(title)$
  <header>
    <h1 class="title">$title</h1>
  $if(author)$
    <p class="author">$for(author)$author$$sep$, $endfor</p>
  $endif$
  $if(date)$
```

```

    <p class="date">${date}</p>
$endif$
</header>
$endif$
<main>
$body$
</main>
</body>
</html>

```

Save this as `my-template.html` and use it with:

```
pandoc input.md --template=my-template.html -t html -o output.html
```

The `sep` in the author loop is a separator that appears between list items but not after the last one — useful for comma-separated author lists.

For LaTeX, a useful minimal template that strips away everything except what you actually need is the starting point for many projects that want full control over the preamble. Start with `pandoc --print-default-template=latex`, save it, and begin removing or adjusting the parts you do not need.

When working with templates, store them in Pandoc's user data directory:

```

# Find your user data directory
pandoc --version | grep "User data"
# Typically: ~/.local/share/pandoc/

# Place templates here for automatic discovery
mkdir -p ~/.local/share/pandoc/templates/
cp my-template.html ~/.local/share/pandoc/templates/

```

Templates placed in the user data directory can be referenced by name without a full path:

```
pandoc input.md --template=my-template -t html -o output.html
```

Defaults files

A defaults file is a YAML file that can specify any Pandoc option that could be passed on the command line. It is the mechanism for creating reusable build configurations without writing shell scripts.

```
# book.yaml
from: markdown
to: pdf
pdf-engine: xelatex
standalone: true
toc: true
toc-depth: 2
number-sections: true
filter:
  - pandoc-crossref
citeproc: true
highlight-style: monochrome
template: book-template.latex
metadata:
  documentclass: book
  classoption:
    - 12pt
    - twoside
    - openright
  mainfont: "EB Garamond"
  monofont: "JetBrains Mono"
  geometry: "margin=25mm, bindingoffset=12mm"
  bibliography: references.bib
  csl: chicago-notes.csl
```

Invoke it with:

```
pandoc --defaults=book.yaml metadata.yaml chapters/*.md -o book.pdf
```

Defaults files can inherit from each other using the `defaults` key, which lets you build a hierarchy — a base configuration extended by format-specific configurations:

```
# html.yaml - extends base settings for HTML output
defaults: base.yaml
to: html5
standalone: true
toc: false
highlight-style: tango
template: web-template.html
output-file: index.html
```

The `output-file` key in a defaults file specifies the output path, which means you can trigger a complete build with nothing but:

```
pandoc --defaults=html.yaml metadata.yaml chapters/*.md
```

This is the simplest possible build command for a complex document. The Makefile patterns in Chapter 10 build on this.

Lua filters

Lua filters are the most powerful extension mechanism Pandoc provides, and they are worth understanding even if you never write one yourself — because Pandoc’s growing ecosystem of pre-written filters provides solutions to a wide range of problems that the base tool does not handle out of the box.

A Lua filter is a Lua script that defines functions named after AST element types. When Pandoc processes a document and encounters an element of that type, it calls the corresponding function, passing the element as an argument. Whatever the function returns replaces the element in the AST.

```
-- uppercase-headings.lua
-- Transform all heading text to uppercase

function Header(e1)
  return pandoc.walk_block(e1, {
    Str = function(str)
```

```

    return pandoc.Str(str.text:upper())
  end
})
end

```

```
pandoc input.md --lua-filter=uppercase-headings.lua -o output.pdf
```

The filter walks every Header element in the AST, and within each header walks every Str inline element, replacing the string text with its uppercase version. The rest of the document is unaffected.

A more practical example — replacing straight quotes with typographic quotes in contexts where Pandoc’s built-in smart quotes handling is insufficient, or converting fenced divs with a specific class to LaTeX environments:

```

-- div-to-env.lua
-- Convert fenced divs with class "theorem" to LaTeX theorem environments

function Div(el)
  if el.classes:includes("theorem") then
    local title = el.attributes.title or ""
    local begin_env = pandoc.RawBlock("latex",
      "\\begin{theorem}" .. (title ~= "" and "[" .. title .. "]" or ""))
    local end_env = pandoc.RawBlock("latex", "\\end{theorem}")
    local content = {begin_env}
    for _, block in ipairs(el.content) do
      table.insert(content, block)
    end
    table.insert(content, end_env)
    return content
  end
end
end

```

With this filter, a fenced div in Markdown:

```

::: {.theorem title="Euclid's theorem"}
There are infinitely many prime numbers.
:::

```

Becomes, in LaTeX output:

```
\begin{theorem}[Euclid's theorem]
There are infinitely many prime numbers.
\end{theorem}
```

And renders correctly in HTML output as a styled `<div class="theorem">` element, because the filter only emits raw LaTeX when the output format is LaTeX.

The key to output-format-aware filters is checking `FORMAT`:

```
function Div(el)
  if el.classes:includes("theorem") then
    if FORMAT == "latex" or FORMAT == "pdf" then
      -- emit raw LaTeX
    else
      -- pass through as a div with appropriate attributes
      return el
    end
  end
end
```

Pandoc's Lua filter API exposes the complete AST through a well-documented library of constructors and accessors. The full API reference is at pandoc.org/lua-filters.html. Commonly useful functions include `pandoc.walk_block`, `pandoc.walk_inline`, `pandoc.RawBlock`, `pandoc.RawInline`, `pandoc.stringify` (converts any AST element to plain text), and `pandoc.read` (parses a string as Markdown and returns an AST, enabling filters that generate document content programmatically).

Multiple filters can be chained by specifying them in sequence:

```
pandoc input.md \
  --lua-filter=filter-one.lua \
  --lua-filter=filter-two.lua \
  --filter=pandoc-crossref \
  -o output.pdf
```

Filters are applied in the order specified. Note that `--filter` (without `lua-`) is for external filter programs that communicate with Pandoc via JSON over standard input/output — an older mechanism that predates Lua filters. Lua filters are preferred for new work because they run in-process and are substantially faster.

Practical conversion patterns

Several conversion tasks come up often enough to be worth treating as patterns.

Converting a legacy LaTeX document to Markdown for future Pandoc-based processing:

```
pandoc legacy.tex -f latex -t markdown -o legacy.md
```

The conversion is imperfect — complex LaTeX commands have no Markdown equivalent and are either approximated or lost — but for documents that are primarily text with standard structural markup, the result is usable and often requires only minor cleanup.

Extracting plain text from a formatted document, for indexing or analysis:

```
pandoc document.docx -t plain -o document.txt
pandoc document.pdf -f pdf -t plain 2>/dev/null # limited support
```

Batch converting a directory of Markdown files to HTML:

```
for f in *.md; do
  pandoc "$f" -t html --standalone -o "${f%.md}.html"
done
```

Or with GNU parallel for speed:

```
ls *.md | parallel pandoc {} -t html --standalone -o {}.html
```

Producing multiple output formats from the same source in a single pipeline:

```
pandoc input.md -o output.pdf --pdf-engine=xelatex &  
pandoc input.md -o output.html --standalone &  
pandoc input.md -o output.epub &  
wait
```

The `&` runs each conversion in the background; `wait` blocks until all three complete. This is a simple form of parallel build that works well for a single document. The Makefile patterns in Chapter 10 extend this to multi-file projects with dependency tracking.

Normalising Markdown from multiple sources to a consistent Pandoc Markdown dialect:

```
pandoc input.md -f gfm -t markdown -o normalised.md
```

This converts GitHub Flavored Markdown to Pandoc Markdown, which is useful when incorporating content from GitHub wikis or README files into a larger document project.

Pandoc's power lies in the combination of its format coverage, its AST-based model, and its extensibility through Lua filters. Most of the document production workflows in Part IV of this book run through Pandoc at some stage — as the primary converter, as a pre-processor that feeds LaTeX, or as a post-processor that cleans up intermediate output. Understanding Pandoc well is, practically speaking, the single highest-leverage skill in CLI typesetting.

The next two chapters narrow the focus to Pandoc's two most important output formats: HTML in Chapter 7, and print-ready PDF in Chapter 8.

Generating HTML

HTML is the most widely read document format in existence. More text is consumed through HTML than through any other medium — more than print, more than PDF, more than any proprietary format. For the CLI typographer, this means that HTML output deserves the same seriousness of purpose as print output: the same attention to typeface selection, the same care about spacing and rhythm, the same respect for the reader's experience.

Pandoc's HTML output is competent by default and excellent with customisation. This chapter covers both: the structures Pandoc produces without intervention, the CSS required to make those structures typographically sound, and the mechanisms for producing single-file or multi-page HTML documents from the same Markdown source.

What Pandoc's HTML output looks like

When you run:

```
pandoc input.md -t html --standalone -o output.html
```

Pandoc generates a complete HTML₅ document with semantic markup. A section heading becomes `<h1>` through `<h6>`. Emphasis becomes ``. Strong becomes ``. Block quotations become `<blockquote>`. Code blocks become `<pre><code class="language-python">`. The document title, if present in the metadata, appears in a `<header id="title-block-header">` element. A table of contents, if requested, appears in `<nav id="TOC" role="doc-toc">`.

This semantic structure is not incidental — it is Pandoc’s most important contribution to HTML output. The alternative, used by many word processors and export tools, is to produce HTML that approximates the visual appearance of the source document using explicit font sizes, inline styles, and layout divs that convey no structural meaning. Pandoc’s output conveys structural meaning first; appearance is a CSS concern.

A compact example. Given this Markdown:

```
---
title: "Font Management from the CLI"
author: "A. N. Author"
date: "March 2024"
lang: en-GB
abstract: |
  An introduction to fontconfig and related tools.
keywords: [fonts, CLI, fontconfig]
---

# Introduction

The `fc-list` command enumerates all fonts known to fontconfig.

## Installation

Install fonts to `~/local/share/fonts/` for per-user access.
```

Pandoc produces:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-GB" xml:lang="en-GB">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <meta name="author" content="A. N. Author" />
  <meta name="keywords" content="fonts, CLI, fontconfig" />
  <title>Font Management from the CLI</title>
  <style> /* default styles */ </style>
</head>
<body>
<header id="title-block-header">
  <h1 class="title">Font Management from the CLI</h1>
```

```
<p class="author">A. N. Author</p>
<p class="date">March 2024</p>
<div class="abstract">
  <div class="abstract-title">Abstract</div>
  <p>An introduction to fontconfig and related tools.</p>
</div>
</header>
<h1 id="introduction">Introduction</h1>
<p>The <code>fc-list</code> command enumerates all fonts known to
  ↳ fontconfig.</p>
<h2 id="installation">Installation</h2>
<p>Install fonts to <code>~/.local/share/fonts</code> for per-user
  ↳ access.</p>
</body>
</html>
```

Several details are worth noting. The `lang` attribute is set from the YAML metadata, which is important for correct hyphenation in browsers and for screen reader behaviour. Keywords become `<meta name="keywords">`. The abstract appears in a `<div class="abstract">` with a titled inner div. Every heading receives an `id` attribute derived from its text, enabling both the table of contents and direct linking to sections.

The default stylesheet Pandoc embeds — accessible via `--print-default-template=html` and visible between the `<style>` tags — is minimal but sensible: it sets a comfortable maximum width of 36em, centres the body, enables hyphenation, applies `text-rendering: optimizeLegibility` and `font-kerning: normal`, and defines basic table and code styles. It deliberately sets no typeface and no font sizes beyond a few responsive adjustments, leaving those decisions to a custom stylesheet.

Semantic HTML and accessibility

Well-structured HTML is simultaneously good typography and good accessibility — two concerns that are more aligned than they might appear.

A document with clear heading hierarchy, properly marked emphasis, labelled tables, and descriptive image alt text serves both the sighted reader scanning for structure and the screen reader user navigating by heading. These are not separate requirements; they are the same requirement: make structure visible.

Several specific points apply to Pandoc-generated HTML.

Heading hierarchy should be continuous. Do not skip from `<h1>` to `<h3>` because you want a visually smaller heading — adjust the CSS instead. Screen readers and assistive technologies use heading levels to build a document outline; gaps in the hierarchy produce confusing navigation. Pandoc's default behaviour produces the right structure from well-organised Markdown; the risk comes from overriding it. If you use a custom template that wraps content in additional elements, ensure the heading levels remain continuous.

Image alt text in Pandoc comes from the Markdown alt text attribute — the text in square brackets before the parenthesised path. For meaningful images, write descriptive alt text that conveys the image's informational content:

```
![Bar chart showing CLI tool adoption rates by year,  
↪ 2015-2024](adoption-chart.png)
```

For purely decorative images, an empty alt attribute tells screen readers to skip the element. In Pandoc Markdown, you can achieve this with an empty alt attribute and a `role` via the attribute syntax:

```
![][decorative-rule.svg]{role="presentation" alt=""}
```

Note that Pandoc wraps images in `<figure>` with the alt text also used as a `<figcaption>`. If this duplication is undesirable for decorative images, a Lua filter can suppress the caption for images with empty alt text.

Tables should use `<thead>` and `<tbody>` correctly, which Pandoc's output does by default. The first row of a Pandoc Markdown table (the row above the separator) becomes `<thead>`; subsequent rows become

<tbody>. Column headers carry the <th> element, which browsers and screen readers recognise as header cells. For complex tables with row headers as well as column headers, Pandoc's table syntax is insufficient and raw HTML is required; this is a known limitation.

Language declaration via the `lang` metadata field should always be set. It affects hyphenation algorithms in browsers, translation tools, screen reader language selection, and the validity of the document under WCAG guidelines. For documents in British English: `lang: en-GB`. For American English: `lang: en`. For documents with sections in multiple languages, the `lang` attribute can be applied to specific elements via Pandoc's span syntax: `[Ce paragraphe est en français.]{lang=fr}`.

Colour contrast is a concern that CSS controls, not Pandoc. Any colour scheme you apply via a custom stylesheet should meet WCAG AA contrast ratios: at minimum 4.5:1 for body text and 3:1 for large text (above approximately 18pt or 14pt bold). For text on a white background, the hexadecimal value #767676 is approximately the minimum grey that meets AA for body text. Values darker than that pass; lighter values fail. Free online tools (WebAIM's Contrast Checker is the standard reference) verify ratios given two hex colour values.

CSS for web typography

Pandoc's default styles are a foundation, not a design. For any document intended to be read seriously — documentation, articles, reports, books — a custom stylesheet is essential. This section builds one from scratch, explaining each decision.

The stylesheet is applied to Pandoc output via the `--css` flag or the `css` metadata variable:

```
pandoc input.md -t html --standalone --css=typography.css -o output.html
```

Or in the YAML metadata block:

```
css: typography.css
```

Multiple CSS files can be specified; they are linked in the order given.

Custom properties establish the design tokens that the rest of the stylesheet references. Define them at the `:root` level so they are available everywhere:

```
:root {
  --font-body: 'EB Garamond', Georgia, serif;
  --font-sans: 'Fira Sans', system-ui, sans-serif;
  --font-mono: 'JetBrains Mono', 'Courier New', monospace;

  --scale-base: 1.2rem;
  --scale-sm: 0.875rem;
  --scale-lg: 1.5rem;
  --scale-xl: 2rem;
  --scale-2xl: 2.75rem;

  --leading: 1.6;
  --measure: 66ch;

  --color-text: #1c1c1c;
  --color-muted: #555;
  --color-link: #1a4e8c;
  --color-code: #f4f4f4;
  --color-border: #ddd;
}
```

The `--measure` variable — `66ch`, approximately 66 characters — enforces the line length principle from Chapter 2 directly in CSS. The `ch` unit equals the width of the `0` character in the current font, which makes it a reliable proxy for average character width.

Web fonts are loaded with `@font-face` for self-hosted fonts or `@import` for Google Fonts. Self-hosting is preferable for privacy (no third-party network request) and performance (no DNS lookup or connection overhead):

```

@font-face {
  font-family: 'EB Garamond';
  src: url('fonts/EBGaramond-Regular.woff2') format('woff2');
  font-weight: 400;
  font-style: normal;
  font-display: swap;
}

@font-face {
  font-family: 'EB Garamond';
  src: url('fonts/EBGaramond-Italic.woff2') format('woff2');
  font-weight: 400;
  font-style: italic;
  font-display: swap;
}

```

`font-display: swap` instructs the browser to render text immediately in a fallback font while the web font loads, then swap in the web font when available. This prevents the *flash of invisible text* (FOIT) that occurs when text is hidden during font loading. The alternative value `font-display: optional` tells the browser to use the web font only if it loads quickly (within a short timeout) and otherwise use the fallback — a reasonable choice for non-essential display fonts.

For Google Fonts, the `@import` URL includes font-display control:

```

@import url('https://fonts.googleapis.com/css2?family=EB+Garamond:ital,wght@0,400;0,500;1,400&family=Fira+Sans:wght@400;500&family=JetBrains+Mono&display=swap');

```

The `display=swap` parameter in the Google Fonts URL applies `font-display: swap` to all loaded faces.

OpenType features relevant to typography should be enabled explicitly:

```

body {
  font-family: var(--font-body);
  font-size: var(--scale-base);
  line-height: var(--leading);
}

```

```
color: var(--color-text);
max-width: var(--measure);
margin: 0 auto;
padding: 2rem 1.5rem 4rem;

/* Rendering quality */
-webkit-font-smoothing: antialiased;
font-kerning: normal;
font-variant-ligatures: common-ligatures;
font-feature-settings: "liga" 1, "kern" 1, "onum" 1;
text-rendering: optimizeLegibility;

/* Hyphenation */
hyphens: auto;
hyphenate-limit-chars: 6 3 3;
overflow-wrap: break-word;
}
```

The `font-feature-settings` declaration enables common ligatures (`liga`), kerning (`kern`), and old-style figures (`onum`). The `onum` feature, where supported by the font, replaces lining figures with text figures that sit more naturally within lowercase text.

`hyphens: auto` enables automatic hyphenation using the browser's built-in hyphenation dictionary for the document's language (set via the `lang` attribute). Without hyphenation, justified text creates ugly word-spacing rivers and ragged-right text produces uneven rags. The `hyphenate-limit-chars` property prevents very short words or fragments from being hyphenated: `6 3 3` means only words of at least 6 characters can be hyphenated, with at least 3 characters before and 3 after the break.

The heading scale uses the sans-serif family for contrast with the body serif, creates a clear size hierarchy, and uses `margin-top` for vertical breathing room:

```
h1, h2, h3, h4, h5, h6 {
  font-family: var(--font-sans);
  font-weight: 500;
  line-height: 1.2;
  color: var(--color-text);
}
```

```

}

h1 { font-size: var(--scale-2x1); margin: 0 0 1rem; }
h2 { font-size: var(--scale-x1); margin: 2.5rem 0 0.75rem; }
h3 { font-size: var(--scale-1g); margin: 2rem 0 0.5rem; }
h4 { font-size: var(--scale-base); font-style: italic; margin: 1.5rem 0
↪ 0.25rem; }

```

Paragraph spacing and indentation choices depend on convention. The British and European convention uses paragraph spacing (a blank line between paragraphs, no first-line indent) for the first paragraph and after headings, and first-line indentation for subsequent paragraphs. This is typographically classical and works well for long-form text:

```

p {
  margin: 0 0 0;
}

p + p {
  text-indent: 1.5em;
}

h1 + p, h2 + p, h3 + p, h4 + p,
blockquote + p, figure + p, .no-indent {
  text-indent: 0;
}

```

The simpler American convention uses paragraph spacing throughout with no indentation — more common in web design:

```

p {
  margin: 0 0 1em;
}

```

Choose one and be consistent.

Code blocks deserve careful styling. The monospace typeface should harmonise with the body type in weight and colour; the background should be subtle:

```
pre, code {
  font-family: var(--font-mono);
  font-size: 0.875em;
  font-feature-settings: normal;
}

pre {
  background: var(--color-code);
  border-left: 3px solid var(--color-border);
  padding: 1rem 1.25rem;
  overflow-x: auto;
  line-height: 1.5;
}

code {
  background: var(--color-code);
  padding: 0.1em 0.3em;
  border-radius: 3px;
}

pre code {
  background: none;
  padding: 0;
}
```

The `font-feature-settings: normal` on code elements resets the OpenType features applied to the body — in particular, ligatures should be off in code, because `fi`, `fl`, and other ligatures in a programming context are confusing and wrong. Many monospace fonts used for code include programming ligatures (ligatures for `->`, `=>`, `!=`, and similar) enabled via their own feature tags; these are a matter of preference and should be enabled deliberately rather than inheriting from the body setting.

Block quotations are understated by default in most browsers. A typographically careful stylesheet sets them off with a vertical marker and appropriate indentation:

```
blockquote {
  margin: 1.5rem 0;
  padding: 0 0 0 1.5rem;
  border-left: 3px solid var(--color-border);
  color: var(--color-muted);
}
```

```

    font-style: italic;
}

blockquote p {
    text-indent: 0;
}

```

Responsive adjustments ensure the document reads well on small screens. The measure (line length) is inherently responsive because it is set in ch units — as the font size changes, the measure changes proportionally. However, padding and font sizes may need explicit adjustment:

```

@media (max-width: 640px) {
    html { font-size: 16px; }
    body { padding: 1rem; }
    h1 { font-size: 2rem; }
}

```

Print styles for HTML documents that may be printed deserve their own @media print block. This is where the web and print traditions converge: you can provide a reasonable printed version of an HTML document without a separate PDF workflow:

```

@media print {
    html { font-size: 11pt; }

    body {
        max-width: none;
        margin: 0;
        padding: 0;
        color: black;
    }

    a { color: black; text-decoration: none; }
    a[href^="http"]:after { content: " (" attr(href) ")"; font-size: 0.8em;
↵ }

    h2, h3 { page-break-after: avoid; }
    p, li { orphans: 3; widows: 3; }
}

```

```
pre { white-space: pre-wrap; page-break-inside: avoid; }  
}
```

The `a[href^="http"]:after` rule prints the URL after each external link — useful for readers who print a document and need to follow a reference.

Syntax highlighting

Pandoc performs syntax highlighting for fenced code blocks at conversion time, producing HTML with `` elements wrapping each token type. Each span carries a class name corresponding to the token category — keywords, strings, comments, operators, and so on — and a stylesheet provides the colours.

The built-in highlight styles are embedded directly in the output HTML as inline `<style>` rules when `--standalone` is used. To use a highlight style with an external CSS file instead — which avoids duplicating the highlighting rules across multiple pages — export the style to JSON, convert it to CSS, and reference it as a stylesheet:

```
# Export the built-in pygments style as JSON  
pandoc --print-highlight-style=pygments > pygments.json  
  
# Use a custom highlight theme from JSON  
pandoc input.md -t html --standalone \  
  --highlight-style=pygments.json \  
  --no-highlight ...
```

Alternatively, suppress Pandoc's highlighting entirely with `--no-highlight` and apply your own CSS. When `--no-highlight` is used, code blocks are output as plain `<pre><code class="language-python">` elements — the same structure used by popular standalone JavaScript highlighting libraries like `highlight.js` and `Prism.js`, which can be loaded via a `<script>` tag:

```

<!-- In your custom HTML template or --include-in-header file -->
<link rel="stylesheet" href=
  ↪ "https://cdnjs.cloudflare.com/ajax/libs/highlight.js/11.9.0/styles/github.min.css"
  ↪ >
<script src=
  ↪ "https://cdnjs.cloudflare.com/ajax/libs/highlight.js/11.9.0/highlight.min.js"
  ↪ >></script>
<script>hljs.highlightAll();</script>

```

The advantage of client-side highlighting is that it supports more languages and more styles than Pandoc's built-in system. The disadvantage is a dependency on JavaScript and a flash of unstyled code during load. For documentation intended to be read online, either approach works; for documents that must degrade gracefully without JavaScript, Pandoc's built-in server-side highlighting is preferable.

For print-oriented documents where colour highlighting is inappropriate (and which may be printed in greyscale), use the monochrome highlight style, which distinguishes token types by weight and style rather than colour:

```
pandoc input.md -o output.html --highlight-style=monochrome
```

Tables of contents

A table of contents in Pandoc HTML output is a `<nav>` element with `role="doc-toc"` containing a nested unordered list of links to each heading:

```
pandoc input.md -t html --standalone --toc --toc-depth=3 -o output.html
```

By default, the TOC appears immediately before the document body. Its position can be controlled via a custom template — you might want it in a sidebar, in a fixed navigation panel, or at the end of the page for short documents. The default template places it after the title block and before the first heading.

The TOC is inserted at the `$table-of-contents$` variable in the template. To move it to a sidebar in a custom template:

```
<div class="layout">
  <aside class="sidebar">
    $if(toc)$
      <nav id="TOC" role="doc-toc">
        $if(toc-title)$<h2>$toc-title$</h2>$endif$
        $table-of-contents$
      </nav>
    $endif$
  </aside>
  <main>$body$</main>
</div>
```

The CSS for a fixed sidebar that remains visible while scrolling:

```
.layout {
  display: flex;
  gap: 3rem;
  max-width: 80rem;
  margin: 0 auto;
  padding: 2rem;
}

.sidebar {
  width: 16rem;
  flex-shrink: 0;
}

.sidebar nav {
  position: sticky;
  top: 2rem;
  font-family: var(--font-sans);
  font-size: 0.875rem;
  line-height: 1.5;
}

.sidebar nav a {
  color: var(--color-muted);
  text-decoration: none;
  display: block;
  padding: 0.15rem 0;
```

```

}

.sidebar nav a:hover { color: var(--color-text); }

main {
  flex: 1;
  min-width: 0; /* prevents flex blowout */
  max-width: var(--measure);
}

```

Active section highlighting — where the TOC entry for the currently visible section is highlighted as the reader scrolls — requires a small amount of JavaScript using the Intersection Observer API. This is beyond Pandoc’s concern but worth adding to any template intended for long-form documentation:

```

const observer = new IntersectionObserver(entries => {
  entries.forEach(entry => {
    const id = entry.target.getAttribute('id');
    const link = document.querySelector(`nav a[href="#${id}"]`);
    if (!link) return;
    link.classList.toggle('active', entry.isIntersecting);
  });
}, { rootMargin: '0px 0px -80% 0px' });

document.querySelectorAll('h1[id], h2[id], h3[id]')
  .forEach(h => observer.observe(h));

```

The `rootMargin` of `-80%` on the bottom means a heading is considered “active” only when it is in the top 20% of the viewport — which typically corresponds to the heading the reader has most recently scrolled past.

Single-file vs multi-page documents

The default Pandoc HTML output is a single file. For most documents — articles, reports, short books — a single file is the right choice: it is simple to deploy, simple to share, and simple to archive.

For long documents — books, documentation sites, multi-chapter references — multi-page output has practical advantages: individual pages load faster, readers can link directly to chapters, and search engine indexing works better when content is divided into semantically coherent units.

Self-contained single files embed all CSS, JavaScript, and images in the HTML file as base64 data URIs, producing a document that requires no external files and can be emailed or archived as a single attachment. The `--embed-resources` flag (combined with `--standalone`) achieves this:

```
pandoc input.md -t html --standalone --embed-resources -o output.html
```

The resulting file can be large — a document with several images may be several megabytes — but it is genuinely self-contained. This is the right format for document delivery: sending a draft to a colleague, archiving a final version, or distributing documentation that must remain intact when its directory structure changes.

Multi-page output with chunkedhtml is Pandoc's built-in multi-page format, introduced in Pandoc 2.17. It splits the document into one HTML file per top-level section and produces a directory containing all generated files plus a `sitemap.json` index:

```
pandoc input.md -t chunkedhtml -o output-directory/
```

The output directory is created by Pandoc; it must not already exist. The generated files follow the naming pattern `1-section-title.html`, `2-next-section.html`, with an `index.html` landing page. Each file contains previous/next navigation links in a `<nav id="sitenav">` element.

The `--split-level` option controls which heading level triggers a page split. The default is 1 (split at every `<h1>`). For a book with parts and chapters, split at level 2 to produce one page per chapter:

```
pandoc input.md -t chunkedhtml --split-level=2 -o output-directory/
```

The `chunkedhtml` format is suitable for documentation sites and books where the content will be served from a web server. For static site generators — Hugo, Jekyll, Eleventy — you will typically generate individual HTML fragments (without `--standalone`) and wrap them in the generator’s own templates, or generate full pages and extract the body content. The right approach depends on the static site generator.

Separate compilation — building each chapter as its own HTML file from a separate Markdown source file — gives maximum control over the output and is the approach most compatible with static site generators:

```
for f in chapters/*.md; do
  name=$(basename "$f" .md)
  pandoc "$f" \
    --defaults=html-defaults.yaml \
    --template=chapter-template.html \
    -o "site/${name}.html"
done
```

In this pattern, each chapter is a standalone HTML page. Navigation between chapters must be handled by the template — typically by reading a JSON sitemap generated separately, or by hardcoding previous/next links in a per-chapter metadata block.

A practical hybrid: generate a single HTML file for local previewing and proofreading (fast, no server required) and a `chunkedhtml` directory for web deployment. A simple Makefile target handles both:

```
CHAPTERS = chapters/01-history.md chapters/02-fundamentals.md ...
DEFAULTS = defaults/html.yaml

preview: output/preview.html
deploy:  output/site/

output/preview.html: $(CHAPTERS)
  pandoc --defaults=$(DEFAULTS) --embed-resources --standalone \
```

```
$(CHAPTERS) -o $@  
output/site/: $(CHAPTERS)  
rm -rf $@  
pandoc --defaults=$(DEFAULTS) -t chunkedhtml \  
$(CHAPTERS) -o $@
```

HTML output from Pandoc is as good as the CSS you bring to it. The default styles are minimal by design — they solve the must-fix problems (line length, hyphenation, kerning, print styles) and leave everything else open. The stylesheet built in this chapter is a solid starting point for any technical document; refine it according to the visual language appropriate to your content and audience.

Chapter 8 turns to the other major output format: print-ready PDF.

Generating Print-Ready PDFs

A PDF intended for professional printing is not simply a PDF that opens in a viewer. It is a document that meets a precise set of technical requirements: fonts fully embedded, colour values in the correct space, bleed and crop marks in position, image resolution adequate for the output dpi, no encryption, no transparency that the RIP cannot process. A PDF that looks perfect on screen but violates any of these requirements may produce unacceptable output on press — or be rejected by the printer outright before it reaches the machine.

This chapter covers the complete pipeline from Markdown source to a PDF you can hand to a printer with confidence: the Typst-backed path, the web-engine path via wkhtmltopdf or WeasyPrint, and the LaTeX path when a legacy class or package stack is unavoidable. It also covers page geometry and margins, headers and footers, font embedding, colour management, and the preflight checks that verify a file is press-ready.

For documents that will be read on screen rather than printed — reports distributed as PDFs, ebooks in PDF format, technical documentation — many of the print-specific requirements (bleed, CMYK, crop marks) do not apply, but font embedding, metadata, and internal link behaviour remain important. The chapter covers both cases.

The three PDF pipelines

Pandoc and Quarto can produce PDF through three fundamentally different routes, each with different strengths.

The **Typst pipeline** converts Markdown to Typst and lets Typst produce the PDF. For many print-oriented projects this is now the cleanest default: it preserves the Markdown-first authoring model, produces strong

PDF output, and avoids pushing the author into TeX macro programming. This is the first route to try for books, reports, and other documents that need deliberate page design without inheriting a full LaTeX stack.

The **LaTeX pipeline** converts Markdown to LaTeX and invokes a TeX engine (pdfLaTeX, XeLaTeX, or LuaLaTeX) to produce the PDF. The TeX engine handles all typesetting: paragraph composition, line breaking, hyphenation, spacing, page layout, footnotes, and everything else. This pipeline remains valuable when a publisher supplies a class file, when specific packages are required, or when an existing house style is already encoded in LaTeX.

The **web-engine pipeline** converts Markdown to HTML and invokes `wkhtmltopdf` (or `WeasyPrint`) to render the HTML page and produce a PDF. The output looks like a browser rendering printed to PDF — which may be exactly what you want if your document is primarily styled in CSS and the visual design is driven by web conventions. This pipeline suits documents that have more in common with web pages than with typeset books: marketing materials, dashboards, browser-rendered reports.

The practical consequence is that these pipelines cannot be mixed casually. You cannot apply TeX's paragraph optimisation to a CSS-styled document, and you cannot apply CSS flexbox layout to a LaTeX document. Choose the pipeline that suits the document type, and apply the appropriate tools within it. In a modern workflow, that usually means Markdown source first, Typst for print-first PDF by default, web-engine PDF when CSS is the real design language, and LaTeX only when its ecosystem is specifically needed.

Page geometry in print-oriented PDF workflows

Every serious print workflow must control the physical dimensions of the page: size, margins, header and footer space, binding offset, and the relationship between the text area and the page boundaries. In Markdown-

first projects, these values belong in metadata or a backend template, not scattered through the manuscript source.

When the backend is LaTeX, the `geometry` package controls these dimensions.

In Pandoc, `geometry` is configured via the `geometry` metadata variable, which is passed directly to the `geometry` package:

```
geometry: "a4paper, margin=25mm"
```

For more precise control, specify dimensions individually:

```
geometry: >  
  a4paper,  
  top=30mm,  
  bottom=25mm,  
  left=30mm,  
  right=25mm
```

For two-sided documents (books, long reports intended for duplex printing), the `twoside` option and `bindingoffset` produce correct inner margins:

```
documentclass: book  
classoption:  
  - 12pt  
  - twoside  
  - openright  
geometry: >  
  a4paper,  
  top=30mm,  
  bottom=25mm,  
  outer=25mm,  
  inner=30mm,  
  bindingoffset=10mm
```

In a two-sided layout, `inner` is the gutter margin (the edge closest to the spine) and `outer` is the fore-edge margin. The `bindingoffset` adds extra

space to the inner margin to account for the binding itself — the amount depends on the binding type and number of pages, but 10–12mm is a typical starting point for perfect binding. The `openright` class option forces each chapter to begin on a right-hand (recto) page, which is standard for professionally typeset books.

Standard page sizes are specified by name in the geometry options. The most common are `a4paper` (210×297mm, the ISO standard used in most of the world), `letterpaper` (8.5×11in, the US standard), `a5paper` (148×210mm, common for smaller books and pamphlets), and `b5paper` (176×250mm, used for some academic publishers' formats). For custom sizes:

```
geometry: "paperwidth=170mm, paperheight=240mm, margin=20mm"
```

The text area — the region of the page that contains body text — is determined by subtracting the margins from the page dimensions. For an A4 page with 25mm margins on all sides, the text area is 160×247mm. For a book with the margins above, it will be smaller. The typographic principle from Chapter 2 applies: the text area should produce a line length of approximately 65–75 characters for comfortable reading. Verify the actual character count with a sample paragraph before finalising the geometry.

The `showframe` option in the geometry package draws visible boxes around the text area, header, footer, and margin notes — essential for checking that your geometry is correct before the document goes to production:

```
geometry: "a4paper, margin=25mm, showframe"
```

Remove `showframe` before the final build.

Headers and footers

Pandoc's default output is intentionally plain. For most professional documents, a custom header or footer is appropriate. In Typst or HTML-based workflows, this belongs in the backend template or stylesheet. In LaTeX, Pandoc's default output uses the `plain` page style, which provides a page number at the bottom centre and nothing else.

The `fancyhdr` package is the standard tool for this. It is configured in the `header-includes` metadata field, which inserts LaTeX commands into the document preamble:

```
header-includes: |
  \usepackage{fancyhdr}
  \pagestyle{fancy}
  \fancyhf{}
  \fancyhead[LE,RO]{\thepage}
  \fancyhead[LO]{\itshape\nouppercase{\rightmark}}
  \fancyhead[RE]{\itshape\nouppercase{\leftmark}}
  \renewcommand{\headrulewidth}{0.4pt}
  \renewcommand{\footrulewidth}{0pt}
```

The position codes are: L (left), C (centre), R (right), combined with E (even/verso pages) or O (odd/recto pages). For single-sided documents, omit the E/O suffix: `\fancyhead[L]`, `\fancyhead[R]`.

The `\leftmark` macro contains the current chapter title (for `book` class) or section title (for `article` class); `\rightmark` contains the current section or subsection title. `\nouppercase` prevents `fancyhdr` from uppercasing these — Pandoc generates headings in sentence case and the default `fancyhdr` uppercasing would be wrong.

For a document with a footer containing the title and page number:

```
header-includes: |
  \usepackage{fancyhdr}
  \pagestyle{fancy}
  \fancyhf{}
  \fancyfoot[L]{\small\textit{The CLI Typographer}}
  \fancyfoot[R]{\small\thepage}
```

```
\renewcommand{\headrulewidth}{0pt}
\renewcommand{\footrulewidth}{0.4pt}
```

The **first page** of a chapter or document typically uses a different style — no header, perhaps a different footer. The `plain` style, which applies to chapter-opening pages by default in the book class, can be customised separately:

```
\fancypagestyle{plain}{%
  \fancyhf{}
  \fancyfoot[C]{\small\thepage}
  \renewcommand{\headrulewidth}{0pt}
}
```

For the title page specifically, the `\thispagestyle{empty}` command suppresses headers and footers entirely. Pandoc applies this automatically to the title block in the `article` class. For book class documents where you want no header or footer on the half-title, title, or copyright pages, place `\thispagestyle{empty}` after those pages' content in the `header-includes` or in a custom template.

Font embedding and verification

A PDF that will be professionally printed must have all fonts embedded. A font that is not embedded is identified by name in the PDF but not defined — the viewer or RIP substitutes a different font when it encounters the name, producing output that differs from the intended design.

In Typst, fonts used in the document are embedded in the generated PDF. In XeLaTeX and LuaLaTeX, all fonts loaded via `fontspec` are automatically embedded and subset in the PDF output. *Subsetting* means that only the glyphs actually used in the document are included in the embedded font data, which reduces file size substantially without affecting fidelity. This is the default and correct behaviour.

In pdfLaTeX, embedding depends on the font package used. Type 1 fonts from well-maintained packages are embedded by default; bitmap fonts (Type 3) from older installations may not be. To verify that all fonts in a PDF are properly embedded, use `pdffonts`:

```
pdffonts output.pdf
```

The output reports, for each font in the document: its name, type, encoding, and whether it is embedded (`emb`) and subsetted (`sub`). A print-ready PDF should show `yes` in both columns for every font:

name	type	emb	sub	uni
-----	-----	---	---	---
APTIGF+DejaVuSerif	CID TrueType	yes	yes	yes
OCVPVM+DejaVuSerif-Bold	CID TrueType	yes	yes	yes
GXKDWJ+DejaVuSansMono	CID TrueType	yes	yes	yes

The random-looking prefix before the font name (APTIGF+, OCVPVM+) is a standard PDF convention indicating that the font is a subset — different documents using the same font will have different prefixes, which is correct behaviour.

If `pdffonts` shows `no` in the `emb` column for any font, that font is not embedded and the PDF is not suitable for professional printing. The fix depends on the engine and font:

- For pdfLaTeX with bitmap Type 3 fonts: install `lmodern` or use a properly packaged Type 1 font. Bitmap fonts appear when LaTeX falls back to Computer Modern bitmap fonts because the scalable (Type 1 or OTF) version is not installed.
- For pdfLaTeX with Type 1 fonts showing `no`: the font's map file may not be properly configured. Run `updmap-sys --enable Map=fontname.map` with the appropriate map file name.
- For XeLaTeX: this should not occur with `fontspec`-loaded fonts. If it does, the font file itself may have embedding restrictions set by the foundry — this is legal information in the font's OS/2 table and XeLaTeX respects it.

The `pdfinfo` tool provides document-level metadata — useful for verifying that the title, author, and other metadata fields were correctly written:

```
pdfinfo output.pdf
```

```
Title:    The CLI Typographer
Author:   A. N. Author
Subject:  Typography and document production
Keywords: typography, CLI, pandoc, Typst
Creator:  Pandoc
Producer: xdvipdfmx (20220710)
Pages:    247
Page size: 595.28 x 841.89 pts (A4)
PDF version: 1.5
```

Pandoc writes the title, author, subject, and keywords metadata to the PDF's document information dictionary through the selected PDF backend. These fields are populated from the YAML metadata block automatically.

PDF metadata and links

Every print-ready PDF still needs correct metadata and predictable link behaviour. In LaTeX output, Pandoc handles this through `hyperref`, which adds navigational hyperlinks to the PDF (from citations to bibliography entries, from the table of contents to sections, from cross-references to their targets) and writes document metadata into the PDF information dictionary.

The most important `hyperref` options for print-ready PDF are controlled through Pandoc metadata variables:

```
colorlinks: true      # colour links instead of boxing them
linkcolor: NavyBlue  # colour for internal links
urlcolor: NavyBlue   # colour for URL links
citecolor: black     # colour for citation links
```

For a PDF destined for print, link colouring should typically be disabled or set to black — coloured links are useful on screen but may appear as unexpected grey patches in greyscale print. Use:

```
colorlinks: false
hidelinks: true
```

`hidelinks: true` removes both the colour and the bounding box that `hyperref` draws around links in its default `colorlinks: false` mode, producing links that are visually invisible but still functional in PDF viewers.

For screen-optimised PDF, coloured links improve navigation. A commonly used palette for professional documents uses `linkcolor: NavyBlue` for internal links and `urlcolor: Maroon` for external URLs — both colours are defined in the `xcolor` package's `dvipsnames` option set, which Pandoc's default template loads.

The `subject` and `keywords` fields in the YAML metadata are written to the PDF as document properties:

```
subject: "A guide to CLI-based typesetting tools and typography"
keywords: [typography, CLI, Pandoc, Typst]
```

These fields are indexed by PDF readers' document search functions and may be used by document management systems. They are not shown in the document body — they are metadata only.

Colour management for print

Print production uses CMYK colour (Chapter 3). Most PDF backends in this book produce RGB output by default, which is acceptable for office printing and screen-viewed PDFs but may not meet the requirements of a professional printing service — particularly for documents with coloured design elements, photographs, or backgrounds.

For documents with no colour beyond black text, this is not a concern: black in RGB (`#000000`) and black in CMYK (`0 0 0 100`) produce identical output, and the printer's workflow will handle the conversion transparently.

For documents with spot colour — a specific accent colour, a brand colour used for headings or rules — the correct approach is to define colours explicitly in CMYK values. The `xcolor` package supports CMYK colour specification:

```
\usepackage[cmyk]{xcolor}
\definecolor{brandblue}{cmyk}{0.85, 0.50, 0.00, 0.10}
```

In a Pandoc document, add this to `header-includes`:

```
header-includes: |
  \usepackage[cmyk]{xcolor}
  \definecolor{accent}{cmyk}{0.85, 0.50, 0.00, 0.10}
```

The `colorspace` package provides more comprehensive CMYK support, including the ability to convert the entire PDF to CMYK output using a specified ICC profile:

```
\usepackage[cmyk, profiles]{colorspace}
```

For most Markdown-sourced documents, the practical approach is simpler: accept RGB output from Typst, LaTeX, or the web-engine pipeline, and if the printer requires CMYK, convert the final PDF using Ghostscript:

```
gs -dBATCH -dNOPAUSE -dNOPROMPT \  
-sDEVICE=pdfwrite \  
-sColorConversionStrategy=CMYK \  
-dProcessColorModel=/DeviceCMYK \  
-sOutputFile=output-cmyk.pdf \  
input-rgb.pdf
```

This Ghostscript command converts all RGB colours in the PDF to CMYK equivalents. The conversion is not always visually perfect — some saturated RGB colours cannot be reproduced in CMYK — but for documents where colour fidelity is not critical, it is a practical solution. For documents where colour fidelity matters (brand materials, photography-heavy publications), work with a proper colour management workflow from the start, with calibrated ICC profiles for both the input and the intended output device.

Bleed and crop marks

Documents printed professionally on sheets larger than the final trimmed size require bleed — content extended beyond the intended trim edge — and crop marks that tell the cutting machine where to trim.

Bleed is typically 3mm on all sides (the printer may specify a different amount). The exact mechanism depends on the PDF backend. In *LaTeX*, use the `geometry` package with a slightly oversized page and the `includeheadfoot` option, combined with the `cropmarks` package or equivalent:

```
geometry: >  
  paperwidth=216mm,  
  paperheight=303mm,  
  top=33mm,  
  bottom=28mm,  
  left=28mm,  
  right=28mm  
header-includes: |  
  \usepackage[a4,center,cam]{crop}
```

Here the paper size is set to A4 plus 3mm bleed on all sides (210+3+3=216mm wide, 297+3+3=303mm tall). The crop package adds camera registration marks (cam) and centres the content on the oversized sheet. The margins include the 3mm bleed.

For most documents distributed as PDF — reports, articles, academic papers, books with white backgrounds — bleed is not needed. It becomes relevant for documents with full-bleed design elements: coloured chapter openers, full-page images, ruled borders that run to the page edge. If your document has none of these, skip bleed entirely.

The geometry package's `showframe` option, used during development, visually confirms that content does not stray into the bleed zone. Only elements intended to bleed should extend past the trim marks.

The wkhtmltopdf pipeline

wkhtmltopdf renders HTML to PDF using the WebKit browser engine. The quality of the result depends on the CSS applied to the source HTML. For well-styled documents, the output is clean and professional; for documents with complex layouts, it may require tuning.

The basic invocation through Pandoc:

```
pandoc input.md -o output.pdf --pdf-engine=wkhtmltopdf
```

Pandoc converts the Markdown to HTML, passes it to wkhtmltopdf, and the PDF is written. By default, the HTML uses Pandoc's default stylesheet; supply a custom CSS file to override it:

```
pandoc input.md -o output.pdf \  
  --pdf-engine=wkhtmltopdf \  
  --css=print.css
```

Margins, page size, and page numbers are controlled via `--pdf-engine-opt`, which passes options directly to wkhtmltopdf:

```
pandoc input.md -o output.pdf \
  --pdf-engine=wkhtmltopdf \
  --pdf-engine-opt="--page-size" --pdf-engine-opt="A4" \
  --pdf-engine-opt="--margin-top" --pdf-engine-opt="25mm" \
  --pdf-engine-opt="--margin-bottom" --pdf-engine-opt="20mm" \
  --pdf-engine-opt="--margin-left" --pdf-engine-opt="25mm" \
  --pdf-engine-opt="--margin-right" --pdf-engine-opt="20mm"
```

For repeated use, these options belong in a defaults file:

```
# wkhtmltopdf-defaults.yaml
from: markdown
to: pdf
pdf-engine: wkhtmltopdf
pdf-engine-opt:
  - --page-size
  - A4
  - --margin-top
  - 25mm
  - --margin-bottom
  - 20mm
  - --margin-left
  - 25mm
  - --margin-right
  - 20mm
  - --enable-local-file-access
```

The `--enable-local-file-access` flag is required when the document references local files (images, local CSS) — `wkhtmltopdf` blocks local file access by default as a security measure.

Page numbers in `wkhtmltopdf` are injected via a header or footer HTML file, not through CSS:

```
# Create a footer HTML file
cat > footer.html << 'EOF'
<!DOCTYPE html>
<html><body>
<div style="font-size: 10pt; text-align: center; width: 100%;">
  <span class="page"></span>
</div>
```

```

</body></html>
EOF

pandoc input.md -o output.pdf \
  --pdf-engine=wkhtmltopdf \
  --pdf-engine-opt="--footer-html" \
  --pdf-engine-opt="footer.html"

```

wkhtmltopdf replaces the class page with the current page number and topage with the total page count — useful for “Page 3 of 12” style footers.

Font embedding in wkhtmltopdf works through web fonts. Any font loaded via `@font-face` in the CSS is embedded in the PDF output. Use `font-display: swap` in the `@font-face` declaration and reference local WOFF2 files for reliable offline builds:

```

@font-face {
  font-family: 'EB Garamond';
  src: url('fonts/EBGaramond-Regular.woff2') format('woff2');
  font-weight: 400;
  font-style: normal;
}

```

Verify font embedding in the output with `pdf fonts`, as described earlier.

Debugging backend-specific PDF failures

When PDF generation fails, diagnose the backend you actually chose. Typst failures are usually simpler and closer to the source. Web-engine failures are usually CSS or asset-path problems. LaTeX failures remain the most opaque, so they deserve special treatment here.

LaTeX errors are notoriously opaque. A missing closing brace five lines earlier produces an error on line 200; a package conflict manifests as

an incomprehensible “dimension too large” message. Some practical strategies.

Read the .log file. When Pandoc reports “Error producing PDF,” it often shows only the last few lines of the LaTeX log. The full log — written to a `.tex.log` file in a temporary directory, or obtainable by adding `--verbose` to the Pandoc invocation — contains the complete error context. Look for the first error (!) in the log, not the last.

Generate intermediate LaTeX. When debugging, stop Pandoc before the PDF stage and inspect the generated `.tex` file:

```
pandoc input.md -t latex -o output.tex
```

Open `output.tex` and compile it manually with your LaTeX engine, which gives better error context than Pandoc’s one-line summary:

```
xelatex output.tex
# or with all error output visible:
xelatex -interaction=errorstopmode output.tex
```

Common errors and their causes:

A ! File ‘X.sty’ not found error means a required package is not installed. Install it via your TeX distribution’s package manager (`tlmgr install packagename` for TeX Live, `miktex-console` for MiKTeX).

A ! Undefined control sequence \X error means either a package providing \X was not loaded, or there is a typo in a header-`includes` LaTeX command. Check the spelling and verify the package is included.

A ! LaTeX Error: Something’s wrong--perhaps a missing \item error in a list environment usually means a list was improperly terminated in the Markdown source — a common cause is a fenced code block inside a list item that was not properly indented.

Overfull \hbox warnings are not errors — they indicate a line that is slightly too wide for the column (extending into the margin). They are

worth investigating in final output: a run of overfull boxes usually signals a long unbreakable word (a URL, a command name) that needs to be handled with `\allowbreak`, `\-soft` hyphen markers, or the `breakurl` package.

Multiple compile passes. Documents with a table of contents, cross-references, or a bibliography require LaTeX to be run more than once — the first run generates auxiliary files (`.aux`, `.toc`, `.bbl`) and the second run reads them to produce correct page numbers and references. Pandoc handles this automatically for simple cases, but complex documents with bibliography processing may need the compile sequence: LaTeX → BibTeX/Biber → LaTeX → LaTeX. The `latexmk` tool automates this:

```
# Generate intermediate .tex, then use latexmk
pandoc input.md -t latex -o output.tex
latexmk -xelatex -interaction=nonstopmode output.tex
```

`latexmk` runs the engine as many times as necessary to resolve all references and produces the final PDF.

A complete print-ready PDF workflow

Combining the elements above, here is a complete Pandoc invocation for a print-ready PDF from Markdown. This example uses Typst as the default PDF backend; switch to LaTeX only when the print workflow requires LaTeX-specific templates or packages:

```
pandoc \
  --defaults=book-defaults.yaml \
  metadata.yaml \
  chapters/01-*.md \
  chapters/02-*.md \
  -o build/output.pdf
```

Where `book-defaults.yaml` contains:

```
from: markdown
to: pdf
pdf-engine: typst
filter:
  - pandoc-crossref
citeproc: true
number-sections: true
toc: true
toc-depth: 2
highlight-style: monochrome
metadata:
  bibliography: references.bib
  csl: chicago-notes.csl
  keywords: [typography, CLI, Pandoc, Typst]
```

If the job requires a LaTeX class or package stack, keep the same Markdown source and switch the defaults file to `pdf-engine: xelatex` plus the required `template:` and LaTeX-specific metadata.

After generating the PDF, run the standard verification checks:

```
# Verify font embedding
pdffonts build/output.pdf | grep -v "^name\|^----"

# Check that all fonts show "yes yes" in emb/sub columns
pdffonts build/output.pdf | awk '$4 != "yes" { print "NOT EMBEDDED:", $0
↵ }'

# Verify document metadata
pdfinfo build/output.pdf | grep -E "^Title:|^Author:|^Pages:|^Page size:"

# Check file size (excessively large PDFs may have unsubsetted fonts)
ls -lh build/output.pdf
```

If all fonts are embedded, the metadata is correct, and the page size matches the intended dimensions, the PDF is ready for print or distribution.

The next chapter turns to a third output format: EPUB, the standard for ebooks. Many of the same source files and Pandoc infrastructure used for PDF apply directly to EPUB output — the differences are in what the format assumes about the reading environment.

EPUB and Ebooks

The ebook is not a digital facsimile of a printed book. It is a different object, with different assumptions about the reading environment. A printed book exists on a fixed canvas — the designer knows the page dimensions, the line length, the typeface, the point size. An ebook exists on a variable canvas that the reader controls: they can change the font, the size, the margins, the background colour, the line spacing. On some devices, they can switch between landscape and portrait orientation. On a phone, the screen may be 320 pixels wide; on a large tablet, 1200 pixels.

The format that embodies these assumptions is EPUB, the open standard for reflowable ebooks. Understanding what EPUB is, how it is structured, and what its constraints mean for typographic decisions is the subject of this chapter. Pandoc produces EPUB output from the same Markdown source used for HTML and PDF — often with minimal additional configuration — but doing so well requires understanding the format’s architecture.

What EPUB is

EPUB is a container format: a ZIP archive with a specific internal structure that a reading system (an ebook reader application or device) knows how to unpack and render. The archive contains XHTML files for the document content, a CSS stylesheet, a manifest listing all the files, a navigation document, and a package document that ties everything together.

Unzip any EPUB and you will find a structure like this:

mimetype	← must be first file, uncompressed
META-INF/	
container.xml	← points to the package document
com.apple.ibooks.display-options.xml	
EPUB/	
content.opf	← package document: manifest + spine
nav.xhtml	← navigation document (EPUB 3)
toc.ncx	← navigation document (EPUB 2, for compatibility)
styles/	
stylesheet1.css	
text/	
title_page.xhtml	
ch001.xhtml	
ch002.xhtml	

The `mimetype` file must contain exactly the string `application/epub+zip` and must be the first file in the archive, stored without compression — this allows reading systems to identify the format by inspecting the first bytes of the file, without fully decompressing it.

The `content.opf` *package document* is the EPUB's manifest and reading order. It lists every file in the publication (in the `<manifest>` element) and specifies the order in which content files should be presented (in the `<spine>` element). A reading system that does not know what else to do with an EPUB can at minimum present the spine items in order.

The `nav.xhtml` *navigation document* is an XHTML file containing the table of contents as a nested `` structure, marked up with `epub:type="toc"`. It is both machine-readable (reading systems use it to populate their navigation panels) and human-readable (it can be presented as a visible table of contents in the document body). The older `toc.ncx` format serves the same purpose for EPUB 2-compatible reading systems; Pandoc generates both.

The content files are XHTML — HTML written to the more strict XML syntax rules. Every element must be properly closed, every attribute must be quoted, and the document must be well-formed XML. Pandoc's EPUB output satisfies these requirements automatically; if you

inject raw HTML into your document via Pandoc's `--include-*` flags or `header-includes`, ensure it is valid XHTML.

Pandoc generates EPUB 3 by default (`-t epub` or `-t epub3`), with EPUB 2 compatibility maintained through the `toc.ncx` file. EPUB 2 output can be requested with `-t epub2`, but there are few reasons to do so: EPUB 2 is a 2007 standard superseded by EPUB 3 in 2011, and all current ebook reading software supports EPUB 3. EPUB 2 output may be necessary only when targeting legacy devices or legacy distribution platforms.

Generating EPUB with Pandoc

The basic EPUB command is:

```
pandoc input.md -o output.epub
```

Pandoc infers EPUB output from the `.epub` extension. The full set of EPUB-specific options appears in the defaults below, which constitute a solid starting point for any EPUB project:

```
# epub-defaults.yaml
from: markdown
to: epub3
split-level: 1
toc: true
toc-depth: 2
epub-title-page: true
metadata:
  lang: en-GB
  toc-title: "Contents"
```

Apply with:

```
pandoc --defaults=epub-defaults.yaml \  
  metadata.yaml \  
  chapters/*.md \  
  -o output.epub
```

`--split-level` determines which heading level triggers a new content file. The default is 1: each `<h1>` heading begins a new XHTML file. For a book structured with parts (`<h1>`) and chapters (`<h2>`), use `--split-level=2` to create one file per chapter. Smaller content files improve navigation performance on some reading systems and are consistent with best practice for long documents.

`--epub-title-page` (true by default) generates an automatic title page from the document metadata. Set to false if you want to supply your own:

```
pandoc --epub-title-page=false ...
```

`--epub-metadata` accepts an XML file containing additional Dublin Core metadata elements that extend the YAML front matter. This is the path to setting metadata that Pandoc's YAML variables do not expose — series information, ISBNs, subject classifications, and accessibility metadata:

```
<!-- epub-metadata.xml -->
<dc:subject
  ↪ xmlns:dc="http://purl.org/dc/elements/1.1/">Typography</dc:subject>
<dc:subject xmlns:dc="http://purl.org/dc/elements/1.1/">Command-line
  ↪ tools</dc:subject>
<dc:rights xmlns:dc="http://purl.org/dc/elements/1.1/">
  Copyright 2024 A. N. Author. All rights reserved.
</dc:rights>
<dc:publisher xmlns:dc="http://purl.org/dc/elements/1.1/"
  ↪ >Self-Published</dc:publisher>
<meta xmlns="http://www.idpf.org/2007/opf"
  property="belongs-to-collection" id="c01">The CLI Series</meta>
<meta xmlns="http://www.idpf.org/2007/opf"
  refines="#c01" property="collection-type">series</meta>
<meta xmlns="http://www.idpf.org/2007/opf"
  refines="#c01" property="group-position">1</meta>
```

```
pandoc input.md --epub-metadata=epub-metadata.xml -o output.epub
```

`--epub-cover-image` specifies a cover image. The image should be a JPEG or PNG, at minimum 1400×2100 pixels at 72 dpi (JPEG is preferred for photographs; PNG for graphics with flat colour). The cover image is displayed by reading systems as the book’s visual identifier — on shelves, in libraries, in search results. A missing or poor-quality cover image significantly affects how the book presents in distribution catalogues:

```
pandoc input.md --epub-cover-image=cover.jpg -o output.epub
```

The cover image is listed in the manifest with the property `cover-image`, which signals to reading systems that it should be used as the visual cover. Pandoc also generates a cover page XHTML file that displays the image full-screen.

EPUB metadata

EPUB metadata is richer than PDF metadata and is used actively by reading systems, library software, and distribution platforms. Getting it right matters more than it might appear.

The essential fields, set in the YAML front matter:

```
title: "The CLI Typographer"
subtitle: "Typography and Typesetting from the Command Line"
author: "A. N. Author"
date: "2024"
lang: en-GB
description: >
  A practical guide to producing high-quality documents
  using command-line tools, covering Pandoc, LaTeX, Typst,
  and the fundamentals of typography.
publisher: "Self-Published"
rights: "Copyright 2024 A. N. Author. All rights reserved."
```

The `lang` field is particularly important — it tells reading systems which language rules to apply for hyphenation, text-to-speech pronunciation,

and other language-sensitive processing. Use a BCP 47 language tag: en for English (generic), en-GB for British English, en-US for American English, de for German, fr for French.

The description field is the book’s blurb — it appears in the EPUB’s metadata and is used by reading systems and distribution platforms that display descriptions. Write it as you would a back-cover description: a few sentences that convey what the book covers and why the reader should read it.

The rights field should contain a copyright statement. For published works, this is typically “Copyright [year] [holder]. All rights reserved.” For open-licensed works, it should state the licence: “This work is licensed under a Creative Commons Attribution 4.0 International Licence.”

The date field should ideally be an ISO 8601 date (2024-03-15) rather than a year alone, as some reading systems use it for sorting. Pandoc accepts either form.

CSS for EPUB

EPUB CSS is HTML CSS, with two significant constraints. First, the reading system controls many properties — it may override your font choices (readers routinely set their own preferred typeface), your font sizes, your line spacing, and your colours (night mode typically inverts the colour scheme). Second, support for CSS properties varies between reading systems; a property that works in every browser may not work in older Kindle firmware or on Sony Reader hardware.

The practical approach is to style what reading systems will not override, be conservative about what you assume the CSS engine supports, and test on representative hardware.

Font stacks should use system fonts as primary choices, with no web fonts loaded from external servers (network requests are often disabled in reading systems). If embedding web fonts, they must be included in the EPUB file itself:

```
body {  
  font-family: Georgia, "Times New Roman", serif;  
  font-size: 1em;      /* never px for body text in EPUB */  
  line-height: 1.6;  
  color: #1a1a1a;  
}
```

Using em units for font sizes is essential. Reading systems allow the user to adjust the base font size; if you set sizes in px, your relative size hierarchy breaks when the user scales the text. Sizes in em or rem scale correctly.

Page breaks in EPUB are controlled by CSS properties that tell the reading system where to force a new screen/page:

```
h1 {  
  page-break-before: always; /* start new page before each chapter */  
}  
  
h2 {  
  page-break-after: avoid; /* try not to break immediately after a  
  ↳ heading */  
}  
  
p {  
  widows: 2;  
  orphans: 2;  
}
```

The `page-break-before: always` on `<h1>` is the most universally important rule for ebooks: it ensures each chapter begins on a new screen rather than running on from the end of the previous chapter. Pandoc's default EPUB stylesheet includes this rule.

First-paragraph indentation in ebooks follows the same convention as print: the first paragraph after a heading has no indent; subsequent paragraphs within a section use a text indent. Some reading systems will fight you on this — particularly Kindle — but the CSS is correct even if it is not universally respected:

```
p {  
  margin: 0;  
  text-indent: 1.5em;  
}  
  
h1 + p, h2 + p, h3 + p, blockquote + p {  
  text-indent: 0;  
}
```

Do not set explicit margins on the body element in EPUB CSS. Reading systems control the page margins through their own settings; overriding them forces your margins on the reader regardless of their preferences. Apply margins only to specific elements that need them within the content area (block quotations, code blocks, figures) rather than to the overall page.

Code blocks in technical EPUBs need careful treatment. Long lines do not reflow in `<pre>` elements — they overflow horizontally. The modern fix:

```
pre {  
  white-space: pre-wrap;  
  word-wrap: break-word;  
  font-family: "Courier New", Courier, monospace;  
  font-size: 0.85em;  
  line-height: 1.4;  
}
```

`white-space: pre-wrap` preserves whitespace and line breaks but allows lines to wrap at the container edge. `word-wrap: break-word` allows wrapping within long tokens (long command names, URLs). This is an imperfect solution — wrapped code is harder to read than formatted code — but it is better than horizontal overflow, which requires the reader to scroll sideways.

Pandoc's default EPUB stylesheet includes this fix as a workaround targeting Apple Books: `@media screen { .sourceCode { overflow: visible !important; white-space: pre-wrap !important; } }`.

Embedding fonts into an EPUB makes it larger but ensures the reading system can use your chosen typeface rather than substituting its default. Font embedding is appropriate when the typeface is integral to the design — a carefully chosen reading typeface, or a monospace font for a technical book where the code appearance matters. Use `--epub-embed-font` to include font files in the EPUB:

```
pandoc input.md \  
  --epub-embed-font=/path/to/EBGaramond-Regular.ttf \  
  --epub-embed-font=/path/to/EBGaramond-Italic.ttf \  
  --epub-embed-font=/path/to/EBGaramond-Bold.ttf \  
  -o output.epub
```

The fonts are copied into an EPUB/`fonts/` directory inside the archive and listed in the manifest. To activate them, reference them via `@font-face` in your CSS:

```
@font-face {  
  font-family: "EB Garamond";  
  src: url("../fonts/EBGaramond-Regular.ttf");  
  font-weight: normal;  
  font-style: normal;  
}  
  
@font-face {  
  font-family: "EB Garamond";  
  src: url("../fonts/EBGaramond-Italic.ttf");  
  font-weight: normal;  
  font-style: italic;  
}  
  
body {  
  font-family: "EB Garamond", Georgia, serif;  
}
```

The META-INF/com.apple.ibooks.display-options.xml file that Pandoc automatically generates contains `<option name="specified-fonts">true</option>`, which tells Apple Books to honour the embedded fonts rather than substituting its own. Other reading systems have

their own mechanisms; Kindle, in particular, requires the obfuscation flag on embedded fonts (more on this below).

Note that embedding fonts increases file size considerably: a typical TTF font file is 200–400KB, and a book with four weights of a typeface can add 1–2MB to the EPUB. For most books this is acceptable. For very large books with already substantial image content, consider using WOFF2 format instead, which is significantly more compressed.

Validation with epubcheck

Before distributing an EPUB — to a retailer, a library platform, or directly to readers — it should be validated with EPUBCheck, the reference EPUB validator maintained by the W3C. EPUBCheck checks conformance with the EPUB 3 specification and reports errors (which will likely cause problems in reading systems) and warnings (which may cause problems in some reading systems).

EPUBCheck is a Java application distributed as a JAR file. Download it from the EPUBCheck releases page on GitHub:

```
# Download EPUBCheck
curl -L
↳ https://github.com/w3c/epubcheck/releases/latest/download/epubcheck.zip
↳ \
  -o epubcheck.zip
unzip epubcheck.zip

# Validate an EPUB
java -jar epubcheck/epubcheck.jar output.epub
```

A valid EPUB produces output like:

```
Epubcheck Version 5.1.0
```

```
Validating using EPUB version 3.3 rules.
No errors or warnings detected.
```

Messages: 0 fatals / 0 errors / 0 warnings / 0 infos

EPUBCheck completed

Common validation errors from Pandoc-generated EPUBs that need manual correction:

Missing or malformed cover image: If `--epub-cover-image` was specified but the image file is missing or is not a supported format, EPUBCheck reports a manifest error. Verify the image path and format.

Raw HTML in content: If you have injected raw HTML via `header-include`s or fenced divs that pass through to HTML, EPUBCheck may report it as malformed XHTML. Ensure all injected HTML is valid XML (closed tags, quoted attributes, no `&` outside of entities).

Undefined epub: type values: If you have manually added `epub:type` attributes to elements, EPUBCheck validates that the values are from the EPUB Structural Semantics Vocabulary. Use only defined values.

Missing language: EPUBCheck warns if no language is set. Always include `lang:` in the YAML front matter.

Pandoc's EPUB output is generally EPUBCheck-clean for standard documents. Errors typically arise from custom templates, injected raw HTML, or cover images with incorrect dimensions or formats.

Reflowable versus fixed-layout EPUB

Every EPUB discussed so far is *reflowable*: the content flows to fit the reading surface, and the reader controls the presentation. This is the correct format for prose — novels, non-fiction, technical books, anything where the text is the primary content and layout is secondary.

Fixed-layout EPUB is a different beast. Each page is a fixed-size canvas, and the content does not reflow. Text, images, and graphic elements are positioned precisely, and the reading system displays them at that size (possibly with pan and zoom on small screens). Fixed-layout EPUB is

appropriate for content where the layout is integral to the work: picture books, comic books, cookbooks with complex two-page spreads, art books, textbooks with tightly integrated figures and text.

Fixed-layout EPUB requires a different authoring approach. Pandoc does not produce fixed-layout EPUB — it is inherently a reflowable tool. Fixed-layout EPUB is typically produced from InDesign, from a purpose-built fixed-layout ebook tool, or by hand-crafting the XHTML with precise CSS positioning. For the CLI typographer, producing fixed-layout EPUB from Markdown is not a realistic workflow; produce a print-ready PDF instead, and consider fixed-layout EPUB only if a reading-system-specific distribution requirement demands it.

Kindle and distribution formats

Amazon's Kindle format is the dominant ebook format for commercial sales. It has gone through several generations: MOBI/PRC (the original format), KF7 (Kindle Format 7, the AZW format on older devices), KF8 (Kindle Format 8, supporting EPUB-like CSS), and KFX (the current format used by Kindle Scribe and high-end devices). All of these are proprietary Amazon formats.

The practical path for producing Kindle content from Pandoc is to produce a clean EPUB 3 and convert it using Calibre's ebook-convert command-line tool:

```
# Convert EPUB to MOBI (Kindle older format)
ebook-convert output.epub output.mobi

# Convert EPUB to AZW3 (Kindle newer format)
ebook-convert output.epub output.azw3

# With metadata options
ebook-convert output.epub output.azw3 \
  --output-profile kindle_pw3 \
  --embed-all-fonts \
  --subset-embedded-fonts
```

Amazon's Kindle Direct Publishing (KDP) platform, the primary channel for self-publishing Kindle books, now accepts EPUB 3 directly and converts it internally. Submitting a well-formed, validated EPUB 3 to KDP is the preferred workflow — it lets Amazon's conversion pipeline optimise for the target device rather than using a double-conversion via Calibre.

If submitting directly as EPUB to KDP, note Amazon's specific requirements that differ from the general EPUB 3 specification:

Font obfuscation is required for embedded fonts in Kindle content submitted to KDP. Obfuscation is a simple XOR encoding applied to the first 1040 bytes of the font file, using the EPUB's unique identifier as the key. The purpose is to prevent casual extraction of the embedded font by someone who unzips the EPUB. Calibre's `ebook-convert` handles obfuscation automatically; if preparing the EPUB by hand, the EPUB obfuscation algorithm is specified in the EPUB 3 standard.

Image requirements: Kindle requires that the cover image be at least 1000×625 pixels (portrait orientation preferred) and recommends 2560×1600 for high-DPI displays. Images within the content should be no larger than 5MB each; very large images should be downscaled.

Table of contents: Kindle requires a navigable table of contents. Pandoc generates both the EPUB 3 `nav.xhtml` and the EPUB 2 `toc.ncx`, both of which KDP accepts.

Apple Books (the Apple distribution platform) accepts EPUB 3 directly and has excellent EPUB 3 support. Apple's distribution tool, `Transporter` (or the `Books` app for self-publishing), validates the EPUB against its own requirements before acceptance. The `com.apple.ibooks.display-options.xml` file that Pandoc generates is specific to Apple Books and enables embedded font support in the Apple Books reading experience.

Draft2Digital, Smashwords, and Ingram are the main aggregator platforms that distribute to retailers other than Amazon. All accept EPUB 3. Draft2Digital converts EPUB to the various retailer-specific formats automatically; supplying a clean, validated EPUB 3 is sufficient.

A practical EPUB build command

Combining the elements above, here is a complete production EPUB build:

```
pandoc \  
  --defaults=epub-defaults.yaml \  
  --epub-cover-image=assets/cover.jpg \  
  --epub-metadata=epub-metadata.xml \  
  --epub-embed-font=fonts/EBGaramond-Regular.ttf \  
  --epub-embed-font=fonts/EBGaramond-Italic.ttf \  
  --epub-embed-font=fonts/EBGaramond-Bold.ttf \  
  --epub-embed-font=fonts/JetBrainsMono-Regular.ttf \  
  --css=styles/epub.css \  
  metadata.yaml \  
  chapters/*.md \  
  -o build/output.epub  
  
# Validate  
java -jar tools/epubcheck.jar build/output.epub  
  
# Convert for Kindle  
ebook-convert build/output.epub build/output.azw3 \  
  --output-profile kindle_pw3
```

Where `epub-defaults.yaml` contains:

```
from: markdown  
to: epub3  
split-level: 1  
toc: true  
toc-depth: 2  
filter:  
  - pandoc-crossref  
citeproc: true  
highlight-style: monochrome  
metadata:  
  lang: en-GB  
  toc-title: "Contents"  
  epub-title-page: true
```

The monochrome highlight style is appropriate for ebooks because coloured syntax highlighting renders poorly in greyscale (the default on e-ink devices) and may be invisible in night mode on LCD devices.

EPUB, PDF, and HTML from the same Markdown source — the three outputs cover the range of how documents are distributed and read. The next chapter builds the automation layer that makes producing all three efficiently from a single build command practical and reliable.

Build Systems and Automation

A document built by hand — running a Pandoc command, checking the output, adjusting flags, running again — is fine for a one-off proof. It is not fine for a book with twenty chapters, three output formats, a bibliography, a custom template, and a production schedule. At that scale, the build process itself becomes a thing to manage: which files are inputs, which are outputs, what depends on what, how to produce all three formats reliably without forgetting a flag, how to know whether the PDF is current or was built from yesterday's source.

These are not new problems. Software engineers solved them fifty years ago. The solution is a *build system* — a tool that knows which outputs need to be rebuilt when inputs change, and that can execute the build steps automatically. The standard Unix build tool, Make, was created in 1976 and remains the right tool for document build pipelines. The same principles that govern compiling a C program — targets, prerequisites, automatic rebuild on change — apply directly to producing PDFs and EPUBs from Markdown.

This chapter covers three tiers: shell scripts for simple cases, Make for serious projects, and CI/CD pipelines for projects where documents must be built automatically on every commit.

Shell scripts for simple builds

The simplest build system is a shell script. When you find yourself typing the same Pandoc command repeatedly, the command belongs in a script.

A minimal build script wraps the Pandoc invocations in a file that can be run with a single command:

```
#!/bin/sh
# build.sh

BOOK="my-document"
META="metadata.yaml"
CHAPTERS="chapters/*.md"
BUILD="build"

mkdir -p "$BUILD"

pandoc --standalone --toc \
  "$META" $CHAPTERS \
  -o "$BUILD/$BOOK.html"

pandoc --toc --split-level=1 \
  "$META" $CHAPTERS \
  -o "$BUILD/$BOOK.epub"

pandoc --pdf-engine=typst --toc \
  "$META" $CHAPTERS \
  -o "$BUILD/$BOOK.pdf"
```

Make it executable and run it:

```
chmod +x build.sh
./build.sh
```

This works. Its limitation is that it rebuilds everything every time — even if you changed only one chapter file, all three formats are rebuilt from scratch. For small documents this is acceptable; for large ones with slow PDF compilation, it becomes painful. This is where Make enters.

Before graduating to Make, however, there is value in hardening the shell script for production use. Three improvements matter most.

Exit on error with `set -e` at the top of the script. Without this, a failing Pandoc command is silently skipped and the script continues, potentially producing stale or empty output files that look valid:

```
#!/bin/sh
set -e # exit immediately if any command fails
```

Prerequisite checking prevents confusing failures downstream:

```
#!/bin/sh
set -e

# Check that required tools are present
for cmd in pandoc tystp; do
  command -v "$cmd" >/dev/null 2>&1 || {
    echo "Error: '$cmd' is required but not installed." >&2
    exit 1
  }
done
```

Timestamped output helps diagnose stale builds when things go wrong:

```
info() { printf "[%s] %s\n" "$(date +%H:%M:%S)" "$*"; }

info "Building HTML..."
pandoc ... -o build/book.html
info "Done: build/book.html ($(du -sh build/book.html | cut -f1))"
```

A complete, production-hardened shell script for a typical book project:

```
#!/bin/sh
# build.sh - production build for a Pandoc book project
set -e

BOOK="cli-typographer"
META="metadata.yaml"
BUILD="build"

die() { echo "Error: $*" >&2; exit 1; }
info() { printf '==> %s\n' "$*"; }

# Prerequisites
```

```

command -v pandoc >/dev/null || die "pandoc not found"
command -v typst >/dev/null || die "typst not found"

CHAPTERS=$(find chapters -name "*.md" | sort)
[ -n "$CHAPTERS" ] || die "No chapter files found in chapters/"

mkdir -p "$BUILD"

info "Building HTML..."
pandoc --defaults=defaults/html.yaml \
  "$META" $CHAPTERS \
  -o "$BUILD/$BOOK.html"

info "Building EPUB..."
pandoc --defaults=defaults/epub.yaml \
  "$META" $CHAPTERS \
  -o "$BUILD/$BOOK.epub"

info "Building PDF..."
pandoc --defaults=defaults/pdf.yaml \
  "$META" $CHAPTERS \
  -o "$BUILD/$BOOK.pdf"

info "Build complete."
ls -lh "$BUILD"/$BOOK.*

```

This script is clear, safe, and correct. What it cannot do is skip formats whose sources have not changed. For that, you need Make.

Make: the right tool for document builds

Make is a build tool that reasons about *targets* (files that need to exist), *prerequisites* (files the target depends on), and *recipes* (the commands that produce the target from its prerequisites). When you run `make`, it checks which targets are older than their prerequisites and rebuilds only those. This is exactly what a document build needs.

A Makefile for a document project has the same structure as a Makefile for a software project, just with Pandoc instead of a C compiler.

The anatomy of a rule

A Make rule has three parts:

```
target: prerequisites
      recipe
```

The recipe line must be indented with a *tab character* — not spaces. This is Make’s most notorious quirk, and it is enforced strictly: a recipe indented with spaces will produce a cryptic error. Every text editor has a setting for this; configure your editor to insert literal tabs in Makefiles, or use the `.editorconfig` file to enforce it across the project.

A minimal rule for building an HTML document:

```
build/book.html: metadata.yaml chapters/01-intro.md chapters/02-body.md
  pandoc --standalone metadata.yaml chapters/01-intro.md \
    chapters/02-body.md -o build/book.html
```

When you run `make build/book.html`, Make checks the modification timestamps of `metadata.yaml`, `chapters/01-intro.md`, and `chapters/02-body.md` against the timestamp of `build/book.html`. If any prerequisite is newer than the target, Make runs the recipe. If the target is already newer than all prerequisites, Make reports `Nothing to be done and exits`.

Variables and automatic prerequisites

Listing every chapter file individually in every rule is impractical for a real project. Make’s variable expansion and wildcard function handle this:

```
CHAPTERS := $(sort $(wildcard chapters/*.md))
```

`wildcard` expands to a space-separated list of all files matching the pattern. `sort` sorts the list and removes duplicates — important because the

chapters must appear in order. The `:=` assignment evaluates the right-hand side immediately, which is the correct behaviour for file lists.

With this variable, the prerequisite list becomes:

```
build/book.html: metadata.yaml $(CHAPTERS) | build
    pandoc --standalone metadata.yaml $(CHAPTERS) -o $@
```

The `$@` automatic variable expands to the target name — `build/book.html` in this case. The `| build` syntax is an *order-only prerequisite*: it ensures the `build/` directory exists before the recipe runs, but a change to the directory itself does not trigger a rebuild.

A complete book Makefile

Here is a production-grade Makefile for a multi-format book project. Read it top to bottom — each section builds on the previous one.

```
# =====
# Configuration
# =====
BOOK      := cli-typographer
CHAPTERS  := $(sort $(wildcard chapters/*.md))
META      := metadata.yaml
BUILD     := build
PANDOC    := pandoc

# Defaults files, one per format
D_HTML    := defaults/html.yaml
D_EPUB    := defaults/epub.yaml
D_PDF     := defaults/pdf.yaml

# Output targets
PDF       := $(BUILD)/$(BOOK).pdf
HTML      := $(BUILD)/$(BOOK).html
EPUB      := $(BUILD)/$(BOOK).epub

# All source files – any change triggers a rebuild
SOURCES   := $(META) $(CHAPTERS)

# =====
```

```

# Phony targets (not files)
# =====
.PHONY: all html epub pdf clean stats watch

all: html epub pdf

html: $(HTML)
epub: $(EPUB)
pdf: $(PDF)

# =====
# Build rules
# =====
$(HTML): $(D_HTML) $(SOURCES) | $(BUILD)
    $(PANDOC) --defaults=$(D_HTML) $(META) $(CHAPTERS) -o $@
    @printf '[HTML] %s (%s)\n' $@ "$$(du -sh $@ | cut -f1)"

$(EPUB): $(D_EPUB) $(SOURCES) | $(BUILD)
    $(PANDOC) --defaults=$(D_EPUB) $(META) $(CHAPTERS) -o $@
    @printf '[EPUB] %s (%s)\n' $@ "$$(du -sh $@ | cut -f1)"

$(PDF): $(D_PDF) $(SOURCES) | $(BUILD)
    $(PANDOC) --defaults=$(D_PDF) $(META) $(CHAPTERS) -o $@
    @printf '[PDF] %s (%s)\n' $@ "$$(du -sh $@ | cut -f1)"

$(BUILD):
    mkdir -p $@

# =====
# Utilities
# =====
clean:
    rm -rf $(BUILD)

# Per-chapter word count with running total
stats:
    @printf '%-40s %8s\n' 'File' 'Words'
    @printf '%-40s %8s\n' '$(shell printf '%0.s-' {1..40})' '-----'
    @total=0; \
    for f in $(CHAPTERS); do \
        n=$$(($(PANDOC) "$$f" -t plain 2>/dev/null | wc -w)); \
        printf '%-40s %8d\n' "$$f" "$$n"; \
        total=$((total + n)); \
    done; \
    printf '%-40s %8s\n' '$(shell printf '%0.s-' {1..40})' '-----'; \

```

```
printf '%-40s %8d\n' 'Total' "$$total"

# Rebuild HTML whenever any source file changes (requires entr)
watch:
  find chapters/ $(META) -type f | entr -s 'make html'
```

Several points about this Makefile are worth explaining.

Defaults files are prerequisites. The HTML rule depends on `$(D_HTML)` as well as the source files. If you change the defaults file — adjusting highlight style, template path, or any other setting — Make will rebuild the HTML. Without this dependency, a change to the defaults file would not trigger a rebuild, and your output would not reflect the new settings.

The @ prefix suppresses command echoing. By default, Make prints each command before running it. Prefixing a recipe line with @ suppresses this — useful for `printf` status lines that replace the raw Pandoc invocation with a cleaner summary. The Pandoc invocation lines do not have @, so you can see exactly what is being run.

The stats target computes word counts in pure Make and shell, avoiding the need for an external script. The double dollar signs (`$$`) are necessary because Make interprets single `$` as its own variable expansion — `$$total` in the recipe becomes `$total` in the shell.

The watch target uses entr, a lightweight file-watching utility (apt install entr on Debian/Ubuntu). It lists all source files, pipes them to entr, and entr re-runs `make html` whenever any listed file changes. This gives you a live-rebuild loop during drafting: save the file, the HTML rebuilds in the background, refresh the browser.

Make in practice

Run `make` with no target to build the default (`all`). Run `make html` to build only HTML. Run `make -n` for a dry run that shows what would be built without building it. Run `make -j2` to run up to two jobs in parallel (useful when building multiple independent formats):

```
make -j2 html epub # build HTML and EPUB in parallel
make -j3 all       # build all three formats, up to 3 at once
```

The `-j` flag is effective for document builds because PDF, HTML, and EPUB compilations are independent — they do not depend on each other’s output. On a machine with multiple cores, `-j3` can reduce total build time by building all three formats simultaneously.

When something goes wrong, `make -n` is the first diagnostic tool: it shows what Make would run without running it, which reveals whether the dependency relationships are correct. `make --debug=basic` goes further, printing Make’s reasoning about which targets are out of date and why.

Version control for prose

Prose belongs in version control for the same reasons code does: to track changes over time, to recover from mistakes, to branch for experiments, and to collaborate. Git is the right tool, and it works on Markdown source just as well as on code — better, in some respects, because Markdown diffs are human-readable.

Setting up a prose repository

The project structure from Chapter 5 maps cleanly to a Git repository:

```
book/
├── .gitignore
├── .editorconfig
├── Makefile
├── metadata.yaml
├── defaults/
│   ├── html.yaml
│   └── epub.yaml
```

```

□   □□□ pdf.yaml
□□□ chapters/
□   □□□ 01-history.md
□   □□□ 02-fundamentals.md
□   □□□ ...
□□□ styles/
□   □□□ epub.css
□   □□□ web.css
□□□ templates/
□   □□□ book.typ
□   □□□ book.latex
□□□ assets/
□   □□□ cover.jpg
□   □□□ figures/
□□□ build/           ← in .gitignore

```

The `build/` directory must be excluded from version control — generated files should not be committed. The `.gitignore`:

```

build/
*.aux
*.log
*.toc
*.out
*.fls
*.fdb_latexmk
*.synctex.gz
.DS_Store

```

The `.editorconfig` enforces consistent formatting across editors and contributors:

```

root = true

[*]
charset = utf-8

```

```
end_of_line = lf
insert_final_newline = true
trim_trailing_whitespace = true

[*.md]
indent_style = space
indent_size = 2
max_line_length = off

[Makefile]
indent_style = tab
indent_size = 4
```

The `max_line_length = off` for Markdown is deliberate: Markdown source benefits from semantic line breaks (one sentence per line, or one clause per line) rather than hard-wrapped lines, because Git diffs are then per-sentence rather than per-paragraph. When you revise a sentence, the diff shows exactly that sentence changed — not a block of reflowed text that obscures what actually changed.

Commit discipline for writing

Git commits in a prose project serve the same purpose as in a code project: they are a record of decisions, a safety net, and a communication tool. The conventions that work well for prose differ slightly from code conventions.

Commit at meaningful stopping points, not after arbitrary numbers of words. A good commit is “complete the section on optical sizing” or “revise the LaTeX engine comparison table” — a unit of editorial work that stands on its own.

Include word counts in commit messages when tracking progress. This creates a useful history of how the manuscript developed:

```
ch03: complete digital typesetting section (~800 words)
```

Covered font formats (OTF/TTF/WOFF), rasterization and hinting,

the print vs screen distinction, and Unicode/OpenType features. Still to do: the summary and transition to ch04.

Use branches for structural experiments. If you are considering a significant reorganisation — splitting a chapter, adding a new section, rewriting a difficult passage — do it on a branch. The main branch stays in a known-good state while you experiment:

```
git checkout -b rewrite/ch02-spacing
# ... work on the chapter ...
git checkout main
git merge rewrite/ch02-spacing # if it worked
# or
git branch -D rewrite/ch02-spacing # if it didn't
```

Tag releases. When a draft reaches a milestone — first complete draft, post-review revision, final submitted version — tag the commit:

```
git tag -a v1.0-first-draft -m "Complete first draft, ~35,000 words"
git tag -a v2.0-final -m "Final version submitted to publisher"
```

Tags let you recover exactly the state of the manuscript at any milestone, which is more reliable than remembering which commit corresponded to which submission.

Diffing prose

Git's default word-diff mode is line-oriented, which is awkward for prose where a single sentence may span multiple lines. The `--word-diff` flag shows changes within lines:

```
git diff --word-diff
```

Output marks deletions with `[-old text-]` and additions with `{+new text+}`:

The `@paper@{+em+}` is a relative unit {+equal to the current font size+}.

For reviewing prose changes, `--word-diff=color` (with a colour terminal) is even clearer:

```
git diff --word-diff=color HEAD~1
```

A Git attribute that improves diffs for Markdown files: tell Git to treat Markdown headers as function names, which makes `git log -p` display the heading context for each change:

```
# .gitattributes
*.md diff=markdown
```

And in `~/.gitconfig` or the repository's `.git/config`:

```
[diff "markdown"]
  xfuncname = "^#{1,6} .*$"
```

With this configuration, `git diff` and `git log -p` show which section each change belongs to, just as they show which function a code change belongs to.

Automating builds with CI/CD

Continuous integration for documents means: every time a commit is pushed, the build runs automatically, and the results — PDFs, EPUBs, HTML — are available without anyone needing to run the build locally. This is particularly valuable for collaborative projects, for books with reviewers who need to access the latest draft, and for documentation that should always reflect the current state of the source.

GitHub Actions

GitHub Actions is the most accessible CI/CD platform for document projects. A workflow file in `.github/workflows/` defines what happens on each push:

```
# .github/workflows/build.yml
name: Build book

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout source
        uses: actions/checkout@v4

      - name: Install Pandoc
        run: |
          wget -q https://github.com/jgm/pandoc/releases/download/3.1.3/\
          pandoc-3.1.3-1-amd64.deb
          sudo dpkg -i pandoc-3.1.3-1-amd64.deb

      - name: Install Typst
        run: |
          curl -fsSL https://typst.community/typst-install/install.sh | sh
          echo "$HOME/.local/bin" >> "$GITHUB_PATH"

      - name: Install fonts
        run: |
          mkdir -p ~/.local/share/fonts
          cp assets/fonts/*.ttf ~/.local/share/fonts/ 2>/dev/null || true
          fc-cache -f

      - name: Build
        run: make all

      - name: Upload build artifacts
```

```

uses: actions/upload-artifact@v4
with:
  name: book-${{ github.sha }}
  path: build/
  retention-days: 90

```

On every push to main, this workflow checks out the repository, installs Pandoc and Typst, builds all formats, and uploads the results as downloadable artifacts. Team members and reviewers can access the latest PDF from the Actions tab without needing to install any tools. If your PDF path still depends on a LaTeX backend, replace the Typst installation step with the relevant TeX packages.

Caching large tool installations dramatically reduces build time. If your workflow still depends on a TeX distribution, caching it avoids reinstalling hundreds of megabytes of packages on every run:

```

- name: Cache LaTeX packages
  uses: actions/cache@v4
  with:
    path: /usr/share/texlive
    key: texlive-${{ runner.os }}-${{
↪ hashFiles('**/texlive-packages.txt') }}

- name: Install LaTeX (if not cached)
  run: |
    sudo apt-get install -y texlive-xetex texlive-fonts-recommended

```

Automatic release publishing on version tags provides polished distribution: when you tag a commit, the CI builds the final versions and publishes them to a GitHub Release page where anyone can download them:

```

- name: Publish release
  if: startsWith(github.ref, 'refs/tags/v')
  uses: softprops/action-gh-release@v1
  with:
    files: |
      build/*.pdf

```

```
build/*.epub
build/*.html
body: |
  Automatically built from ${ github.sha }.
```

A simpler CI with a shell script

Not every project needs GitHub Actions. For a project hosted on any Linux server you control — a VPS, a home server, a shared university server — a simple post-receive Git hook achieves the same result:

```
#!/bin/sh
# .git/hooks/post-receive
# Runs on the server after each push

REPO="$HOME/book-repo"
BUILD_DIR="$HOME/public_html/book-drafts"
WORK_TREE="/tmp/book-build-$$"

mkdir -p "$WORK_TREE"

# Check out the latest commit to a temporary directory
git --work-tree="$WORK_TREE" checkout -f main

# Build
cd "$WORK_TREE"
mkdir -p build
make html epub 2>&1 | tee build/build.log

# Publish
mkdir -p "$BUILD_DIR"
cp build/*.html build/*.epub "$BUILD_DIR/"
cp build/build.log "$BUILD_DIR/"

# Clean up
rm -rf "$WORK_TREE"

echo "Build complete. Files available at $BUILD_DIR"
```

Make the hook executable (`chmod +x .git/hooks/post-receive`) and it will run automatically after every push. The built files appear in

~/public_html/book-drafts/, which if the server runs a web server becomes a URL your collaborators can access.

Putting it together: a complete project scaffold

Combining everything in this chapter, here is the full directory structure and file set for a production document project. This can serve as a template for any new book or long-form document project.

```
book/
  □□□ .editorconfig
  □□□ .gitattributes
  □□□ .gitignore
  □□□ .github/
  □ □□□ workflows/
  □ □□□ build.yml
  □□□ Makefile
  □□□ metadata.yaml
  □□□ defaults/
  □ □□□ html.yaml
  □ □□□ epub.yaml
  □ □□□ pdf.yaml
  □□□ chapters/
  □ □□□ 01-history.md
  □ □□□ 02-fundamentals.md
  □ □□□ ...
  □□□ styles/
  □ □□□ web.css
  □ □□□ epub.css
  □□□ templates/
  □ □□□ book.typ
  □ □□□ book.latex
  □□□ assets/
  □ □□□ cover.jpg
  □ □□□ fonts/
```

```
□ □□ figures/  
□□□ tools/  
    □□□ build.sh          ← fallback for environments without Make
```

To start a new project from this scaffold:

```
# Clone or copy the scaffold  
cp -r book-scaffold/ new-book/  
cd new-book/  
  
# Initialise git  
git init  
git add .  
git commit -m "Initial project scaffold"  
  
# Verify the build works  
make html
```

The scaffold is the lowest-friction starting point for serious document work. The Makefile handles incremental builds. Git handles history and collaboration. The defaults files handle format-specific configuration. The CI workflow handles automatic building and distribution. Together, they make the administrative overhead of a multi-format document project — which can otherwise consume a significant fraction of a writer’s time — nearly invisible.

Part II ends here. We have covered Markdown as a source format, Pandoc as the conversion engine, HTML and PDF and EPUB as outputs, and Make and Git as the infrastructure that makes producing all of them reliable and automatic. Part III turns to the broader toolbox: LaTeX in depth, Typst, Quarto, Emacs and Org Mode, and the Unix heritage tools that predate all of them.

The Toolbox

Choosing the Right Tool

Part III of this book covers six distinct tools or tool families — LaTeX, Typst, Quarto, Emacs and Org Mode, groff and the Unix heritage, and diagram generators. Each of them could be used to produce almost any document. Each of them would be the wrong choice for some documents and the right choice for others. Before examining any of them in depth, it is worth building a framework for deciding which to reach for.

The wrong approach is to pick a tool you know and use it for everything. This works, but it produces documents that fight their tools: LaTeX used for a quick internal memo, Pandoc Markdown used for a paper with hundreds of custom mathematical environments, Typst used for a 500-page book with a complex existing LaTeX infrastructure. Every tool has a domain where it is excellent and a domain where it becomes friction.

The right approach is to understand what each tool is optimised for, what it costs to use it, and what it produces — then match the tool to the document. For this book’s purposes, the default hierarchy is simple: Markdown plus Pandoc or Quarto when you need HTML, EPUB, and PDF from one source; Typst when you need a print-ready PDF from a modern toolchain; LaTeX when an existing ecosystem forces the choice.

Three axes of decision

Most document production decisions resolve along three axes: the *output format* you need, the *complexity* of the document, and the *workflow* you are operating in.

Output format is often the most constraining factor. If you must deliver a DOCX file to a publisher, LaTeX is a poor starting point regardless of its typographic quality. If you are producing a data-driven report where the content changes with each run, a system that cannot execute code is the wrong choice. If you need to produce HTML, PDF, and EPUB from the same source, a tool that only targets one output has already ruled itself out.

Document complexity spans a wide range. A one-page letter and a five-hundred-page technical reference are both “documents,” but they require entirely different levels of infrastructure. A letter needs: a typeface, some spacing, a date, perhaps a salutation. A reference manual needs: a consistent heading hierarchy, automatic cross-references, a generated index, a bibliography, running headers that reflect the current chapter, figure captions numbered by chapter, consistent code formatting across hundreds of examples. Using a book-class LaTeX setup for the letter is overkill; using a simple Markdown-to-HTML pipeline for the reference manual is under-equipped.

Workflow encompasses both the human and technical environment. Who is writing the document? A lone author comfortable with the terminal has different constraints than a team where some members use graphical editors. Is the document produced once or regenerated regularly? Is it part of a larger build pipeline? Does it need to be reproducible — does someone need to rebuild it years later and get the same output? Does it need to integrate with data sources, Jupyter notebooks, or other programmatic inputs?

These three axes interact. A data-driven document (workflow) probably needs HTML and PDF output (output format) and has moderate complexity. That combination points strongly toward Quarto or Pandoc with a scripting layer. A complex mathematical monograph (complexity) needs print-quality PDF (output format) and is probably authored by a single specialist comfortable with technical tools (workflow). That points to Typst first, or to LaTeX where compatibility requirements dominate.

The tools and their niches

Pandoc is a converter and a document assembler. It is not primarily a typesetting system — it does not compute optimal line breaks, it does not handle complex page layout, and it does not provide a composition language of its own. What it provides is breadth: it reads more formats than any other tool, writes more formats than any other tool, and its Lua filter system allows significant customisation of the conversion process. Pandoc's niche is documents where the *source format matters* — where you want to write in Markdown and produce multiple output formats, or where you are working with content that arrives in various formats and needs to be normalised.

Pandoc is the right choice when: you need multiple output formats from one source; you are working with existing Markdown content; you want to stay close to plain text with minimal markup; your document is primarily prose with standard structural elements (headings, lists, tables, citations, footnotes); or you are building a document pipeline that needs to process content programmatically.

Pandoc is the wrong choice when: you need precise control over page layout and typography beyond what LaTeX's default Pandoc template provides; your document has complex custom environments or heavy mathematical content that benefits from native LaTeX authoring; or you need the full programmability of a Turing-complete document language.

LaTeX is the historical standard for print-quality technical typesetting, especially in mathematics and publisher-driven workflows. It is a typesetting system first and a document format second — the author writes markup that instructs the typesetting engine, and the engine makes sophisticated decisions about line breaking, hyphenation, spacing, and page layout. Its ecosystem of packages is vast: virtually any typographic requirement has been addressed by some package somewhere on CTAN.

The cost of LaTeX is friction. Its syntax is verbose. Its error messages are cryptic. Debugging a complex document can require real expertise.

Customising the appearance requires knowing which of the hundreds of relevant packages to use, and learning their individual syntaxes. Getting LaTeX to do something it was not designed to do — a highly designed magazine layout, a document with complex floated elements in precise positions — can require work that a graphical tool would accomplish in minutes.

LaTeX is the right choice when: you are submitting to a publisher or conference that requires LaTeX input; the document depends on a substantial existing LaTeX infrastructure; the mathematical requirements lean on packages that Typst or other systems do not yet match; or a long-standing institutional workflow already assumes LaTeX.

LaTeX is the wrong choice when: you need non-PDF output as a primary format; you are collaborating with people who cannot or will not use LaTeX; the document is short, simple, or primarily visual in design; or the friction cost of LaTeX's learning curve is not justified by the project's requirements.

Typst is a modern document composition system designed to do what LaTeX does while being substantially more approachable. Its syntax is cleaner and more consistent than LaTeX's, its error messages are informative, its compilation is fast (it compiles incrementally, so large documents update in milliseconds rather than seconds), and its scripting language — a proper functional language built into the tool — makes programmatic document generation natural. For new PDF-only projects, it should generally be the first option you evaluate.

Typst is the right choice when: the deliverable is primarily a print-ready PDF; you want high-quality output without inheriting LaTeX's macro language; you are starting a new project without legacy LaTeX infrastructure to integrate; or you need programmable layout in a cleaner language.

Typst's limitations are mainly ecosystem and maturity. It was publicly released in 2023, and its package ecosystem is small compared to CTAN. Some capabilities that LaTeX packages provide — very specific journal templates, certain mathematical environments, niche typographic packages — do not yet have Typst equivalents. For standard documents, these

gaps rarely matter. For documents with highly specific requirements (a paper for a journal with a LaTeX-only submission system, a document that depends on a specific CTAN package), they may be decisive.

Quarto is a scientific and technical publishing system built on Pandoc, with native support for R, Python, Julia, and Observable JavaScript. Its central capability is *literate programming*: weaving executable code with prose narrative in a single source document, so that figures, tables, and computed values are generated fresh each time the document is built. This makes Quarto the right tool for reproducible research — documents where the analysis and the narrative are the same artifact.

Beyond reproducibility, Quarto provides a polished publishing system: a book format with automatic chapter numbering and cross-references, a website format with navigation and search, a presentation format, and a dashboard format, all from the same source syntax (a superset of Pandoc Markdown called QMD). Its templates and themes are well-designed, and for academic use cases it handles citations, cross-references, and bibliography management with less configuration than raw Pandoc.

Quarto is the right choice when: the document is a data analysis, a research report, or any document where content is generated programmatically; you are working in R, Python, or Julia and want the analysis and the document to be the same artifact; you need a polished website or multi-format output with minimal configuration; or you are working in an academic environment where reproducibility is required or valued.

Quarto is the wrong choice when: the document does not involve code execution; you need output formats that Quarto does not support well; or the additional layer between Markdown and Pandoc adds complexity that plain Pandoc would not.

Emacs and Org Mode occupy a different category. Org Mode is a major mode for Emacs — which is itself a programmable text editor — that provides an outline-based document format with an exceptionally rich feature set: task management, spreadsheet-like table editing, literate programming (via Babel, which predates Quarto's code execution by years), export to a wide range of formats, and deep integration with the Emacs

ecosystem. For authors who already live in Emacs, Org Mode is a natural choice for document production.

The caveat is obvious: Org Mode is inseparable from Emacs. The learning curve for Emacs is steep, the tool is deeply opinionated, and the workflow is idiosyncratic by the standards of contemporary software. Org Mode is the right choice if and only if Emacs is already your working environment or you are willing to make it so. It is not worth adopting Emacs for the sake of Org Mode’s document features, because Pandoc, Quarto, Typst, and LaTeX collectively cover the same ground for authors who do not already live in Emacs.

The groff family — troff, nroff, and their derivatives — are the original Unix typesetting tools, created at Bell Labs in the early 1970s. They are still the tools used to format manual pages on Unix and Linux systems. For man page authorship, groff is the correct and standard tool. For anything else, it is a historical artifact worth understanding but not worth adopting for new work.

The comparison matrix

Evaluating these tools across the dimensions that matter for practical document work:

Criterion	Pandoc	LaTeX	Typst	Quarto	Org Mode
Output formats	★★★★★	★★☆☆☆	★★☆☆☆	★★★★★	★★★★☆
Typographic quality	★★★★☆	★★★★★	★★★★★	★★★★☆	★★★★☆
Math support	★★★☆☆	★★★★★	★★★★☆	★★★★☆	★★★★☆
Learning curve	★★★★☆	★★☆☆☆	★★★★☆	★★★★☆	★☆☆☆☆
Plain text authoring	★★★★★	★★★★☆	★★★★☆	★★★★★	★★★★★
Code execution	☆☆☆☆☆	☆☆☆☆☆	☆☆☆☆☆	★★★★★	★★★★★
Ecosystem / packages	★★★★☆	★★★★★	★★☆☆☆	★★★★☆	★★★★☆
Compilation speed	★★★★★	★★☆☆☆	★★★★★	★★★★☆	★★★★☆
Long document support	★★★★☆	★★★★★	★★★★☆	★★★★☆	★★★★☆
Reproducibility	★★★★☆	★★★★★	★★★★☆	★★★★★	★★★★☆

Ratings are relative to the document production use case, not absolute capability.

A few cells deserve commentary.

Pandoc's output format breadth is genuinely unmatched — it writes to over forty formats from any of its input formats. LaTeX's output format score is low not because LaTeX is weak but because it targets PDF almost exclusively; getting other output formats from LaTeX requires conversion pipelines.

LaTeX's typographic quality score reflects the historical quality ceiling and the depth of its ecosystem. Typst receives the same score because for new PDF-only work it achieves comparable results with far less friction. The recommendation difference between them is not about whether one can make beautiful pages and the other cannot; it is about whether you need LaTeX's accumulated compatibility layer.

Quarto's learning curve rating of four stars reflects that for its core use case — computational documents — Quarto abstracts away most of the complexity. For authors already familiar with Pandoc Markdown, Quarto is a modest extension, not a new system to learn.

Org Mode's learning curve rating of one star is not a criticism of Org Mode. It is a reflection of the prerequisite: Emacs. Org Mode itself, once you are in Emacs, is relatively learnable. The combined system of Emacs-plus-Org-Mode is the most demanding starting point of anything in this book.

A decision flowchart

The decision between tools can be stated as a series of questions that progressively narrow the field.

Does the document contain executable code that generates figures, tables, or computed values? If yes: Quarto. Its code execution support is far more mature than any alternative, and its output quality for computational documents is excellent. If no: continue.

Does the document require LaTeX as input format — for a journal submission, a conference proceedings, a publisher with a LaTeX house style?

If yes: LaTeX directly, possibly with Pandoc as a preprocessing stage for the Markdown-to-LaTeX path. If no: continue.

Is the document primarily mathematical — theorems, proofs, substantial notation? If yes, and if you are starting fresh with no legacy constraints: Typst is the first option to evaluate. If you need maximum ecosystem compatibility or are joining an existing mathematical project: LaTeX. If no: continue.

Do you need more than one output format from the same source — for instance, HTML, EPUB, and PDF? If yes: Pandoc or Quarto (Quarto if the document has any computational content; Pandoc for pure prose). If no: continue.

Is the document primarily a print artifact and PDF-only — a report for distribution, a book proof, a designed handout? If yes: Typst first. Choose LaTeX only if legacy classes, package dependencies, or submission requirements make it unavoidable. If no: continue.

Is it a short-to-medium prose document — an article, a report, documentation? Pandoc Markdown with appropriate defaults is the simplest path when multiple formats matter. Typst is the better path when the target is PDF only.

Do you already work in Emacs? If yes, Org Mode is worth serious consideration for any document that fits its export capabilities. If no, it is not worth adopting Emacs for document production alone.

Hybrid workflows

The tools are not mutually exclusive. Several hybrid patterns are common and worth knowing.

Pandoc as front-end for LaTeX. Write in Pandoc Markdown. Use Pandoc to generate LaTeX, either as an intermediate step in the PDF pipeline

or as a file you then hand-edit for final production. This gives you Markdown's clean authoring experience for the bulk of the work, with LaTeX's full power available for the parts that need it. The Lua filter system means you can handle many complex cases without touching the generated LaTeX directly.

Quarto for computation, Typst for PDF by default. Quarto can drive multi-format output while delegating the PDF layer to a dedicated engine. For new work, prefer a Typst PDF backend where the formatting requirements permit it; keep LaTeX in reserve for journals, classes, and package-heavy workflows.

Pandoc for content, Typst for layout. Pandoc's Typst output format (added in Pandoc 3.1) allows you to write Markdown and produce Typst source, which can then be compiled with the Typst engine. The Typst file can be further customised for layout purposes. This is a newer pattern and the Typst output format is still maturing, but it offers a promising path for authors who want Markdown authoring and Typst's output quality.

Make to orchestrate multiple tools. A `Makefile` can invoke Typst for the print PDF, Pandoc for HTML and EPUB, and Quarto for a companion website, with LaTeX retained only where compatibility requires it. The build system is the integration layer that makes the combination tractable.

The practical implication is that you should not feel forced to choose a single tool for an entire project. The right granularity is the output format or the document component: use the best tool for each.

A note on staying current

The landscape of CLI typesetting tools is not static. Typst did not exist before 2023. Quarto replaced R Markdown in 2022. Pandoc adds major features with each release. New tools emerge regularly.

The framework in this chapter — asking about output format requirements, document complexity, and workflow constraints — will remain

useful even as the specific tools change. What changes is which tool best satisfies each combination of requirements. As of this writing, the assessments above reflect the current state of the tools: Typst is the preferred PDF-only path for new work, Pandoc and Quarto dominate the Markdown-to-many-formats workflow, and LaTeX remains strongest where legacy infrastructure is decisive. In two or three years, that balance may shift again.

The deeper skill is not knowing which tool to use today. It is knowing how to evaluate a new tool against the requirements of a specific document — which is what the framework in this chapter equips you to do.

The next four chapters examine each major tool in depth: LaTeX in Chapter 12, Typst in Chapter 13, Quarto in Chapter 14, and Emacs and Org Mode in Chapter 15. Chapter 16 covers groff and the Unix heritage, and Chapter 17 covers diagram tools that complement all of the above.

LaTeX

LaTeX is the historical centre of modern technical typesetting, not the inevitable destination of all serious document work. For new PDF-only projects, this book generally prefers Typst; for multi-format work, it prefers Markdown with Pandoc or Quarto. LaTeX remains important because publishers, conferences, long-lived institutional workflows, and specialised mathematical packages still depend on it. Understanding it deeply, rather than just knowing enough to make Pandoc produce PDFs, pays dividends whenever you hit those compatibility boundaries.

This chapter covers LaTeX as a direct authoring tool: the document structure, the engine choices, the essential packages, the mathematics system, bibliography management, and the techniques for building reusable document classes. It assumes you have a TeX Live installation; if not, the first section explains how to get one.

TeX distributions

LaTeX is not a single program. It is an ecosystem of tools, engines, packages, and fonts distributed as a collection called a *TeX distribution*. The three distributions in current use are:

TeX Live is the standard distribution for Linux and macOS, maintained by the TeX Users Group and updated annually. On Debian and Ubuntu systems, a minimal installation suitable for most work is:

```
sudo apt install texlive-xetex texlive-fonts-recommended \  
texlive-latex-extra texlive-science
```

For a complete installation including all packages, `texlive-full` provides everything at the cost of approximately 5GB of disk space. The targeted installation above is preferable for most users.

MiKTeX is the standard distribution for Windows, with a distinctive feature: it installs missing packages on demand. When a document requires a package that is not installed, MiKTeX downloads and installs it automatically during compilation. This eliminates the need to think about package installation at all, at the cost of internet access during first use of each new package.

TinyTeX is a lightweight LaTeX distribution designed for R and Quarto users, installable without root privileges. Its `tlmgr install` command installs individual packages on demand, like MiKTeX. For authors who want a minimal, reproducible LaTeX installation without a full TeX Live, TinyTeX is a good choice:

```
# Install TinyTeX
curl -sLO https://yihui.org/tinytex/install-unx.sh
sh install-unx.sh
```

Once a distribution is installed, **tlmgr** is the package manager for TeX Live and TinyTeX:

```
tlmgr install biblatex biber # install specific packages
tlmgr update --all          # update all packages
tlmgr search --global cleveref # search for a package
```

texdoc is the documentation viewer. Every CTAN package ships with documentation, and `texdoc` opens it:

```
texdoc geometry # open geometry package documentation
texdoc microtype # open microtype documentation
texdoc fontspec # open fontspec documentation
```

When in doubt about a package option, `texdoc packagename` is the right first resource. The documentation for major packages is detailed, well-written, and authoritative.

Document structure

A LaTeX document has two parts: the *preamble* and the *body*. The preamble begins with `\documentclass` and extends to `\begin{document}`. The body is everything between `\begin{document}` and `\end{document}`.

```
\documentclass[11pt,a4paper]{article} % class and options

% --- Preamble: packages and settings ---
\usepackage[T1]{fontenc}
\usepackage{geometry}
\geometry{margin=25mm}

% --- Body ---
\begin{document}

\section{Introduction}

Body text goes here.

\end{document}
```

The `\documentclass` command specifies the class — the template that defines the overall document structure and appearance — and its options. The most commonly used standard classes are `article` (for papers and short documents), `report` (for longer documents with chapters), `book` (for books with front matter, chapters, and back matter), and `letter`. The KOMA-Script alternatives `scrartcl`, `scrreprt`, and `scrbook` replace these with more flexible, European-oriented equivalents and are the better choice for new documents.

Class options are comma-separated values in square brackets. The most important are:

- `10pt`, `11pt`, `12pt` — base font size (default is `10pt`)
- `a4paper`, `letterpaper`, `a5paper` — paper size
- `twoside`, `oneside` — page layout (`twoside` uses different inner/outer margins for recto/verso pages)

- `openright`, `openany` — whether chapters start on right-hand pages (book and report only)
- `fleqn` — left-align displayed equations instead of centring them
- `leqno` — number equations on the left instead of the right

The preamble loads packages and configures settings. Package loading order matters: some packages must be loaded before others, and some conflict if loaded in the wrong order. The general rule is: load encoding packages first (`fontenc`, `inputenc`), then fonts, then layout packages (`geometry`, `fancyhdr`), then feature packages (`microtype`, `hyperref`), and load `hyperref` last among the feature packages since it modifies many internals.

Encoding in pdfLaTeX requires two declarations:

```
\usepackage[T1]{fontenc}      % output font encoding – use T1 for Western
↪ European
\usepackage[utf8]{inputenc}  % input encoding – use utf8 for modern
↪ systems
```

XeLaTeX and LuaLaTeX assume UTF-8 input and use Unicode font encoding natively; neither `inputenc` nor `fontenc` should be loaded in those engines (or load `fontspec` which handles this correctly).

Engines: pdfLaTeX, XeLaTeX, LuaLaTeX

The three main LaTeX engines produce identical-looking output for standard documents but differ significantly in their font handling and capabilities.

pdfLaTeX compiles `.tex` source directly to PDF. It requires fonts to be available in Type 1 format and configured through LaTeX's font selection system (NFSS). Its main advantage is speed and stability; its main limitation is that it cannot use arbitrary system fonts. Standard LaTeX font packages — `palatino`, `tgpagella`, `newtxtext`, `libertinus`, `mathpazo` — all work with pdfLaTeX.

```
% pdfLaTeX font selection via packages
\usepackage{palatino}           % Palatino for text
\usepackage[sc]{mathpazo}      % Palatino for math, with small caps
\usepackage[scaled=0.9]{helvet} % Helvetica for sans-serif
\usepackage{courier}          % Courier for monospace
```

XeLaTeX uses Unicode natively and accesses any OpenType or TrueType font installed on the system via the `fontspec` package. This is the engine to use for documents that need specific professional typefaces, non-Latin scripts, or advanced OpenType features.

```
% XeLaTeX font selection via fontspec
\usepackage{fontspec}
\setmainfont{EB Garamond}[
  Numbers = OldStyle,
  Ligatures = TeX
]
\setsansfont{Fira Sans}[Scale = MatchLowercase]
\setmonofont{JetBrains Mono}[Scale = MatchLowercase]
```

The `Scale = MatchLowercase` option adjusts the font's size so its x-height matches that of the main font — critical for visual harmony when mixing typefaces.

LuaLaTeX shares XeLaTeX's Unicode and `fontspec` capabilities and adds a full Lua scripting environment accessible from within the document. This enables genuinely programmatic document generation: you can write Lua code that reads external data, performs calculations, and generates LaTeX commands, all within a single `.tex` file.

```
% LuaLaTeX: generate a table from Lua
\directlua{
  local data = {{"Pandoc", "Markdown", "Any"}, {"LaTeX", "TeX", "PDF"}}
  tex.sprint("\begin{tabular}{lll}")
  for _, row in ipairs(data) do
    tex.sprint(table.concat(row, " & ") .. " \\")
  end
  tex.sprint("\end{tabular}")
}
```

Choose pdfLaTeX when you are using standard LaTeX font packages and do not need system fonts. Choose XeLaTeX when you need specific OTF/TTF fonts. Choose LuaLaTeX when you need programmatic document generation or need LuaLaTeX-specific packages.

Mathematics

LaTeX's mathematics typesetting is its most celebrated capability, and the reason it remains the standard for scientific publishing forty years after its creation. The quality of LaTeX mathematics — the spacing around operators, the sizing of delimiters, the placement of subscripts and superscripts, the layout of multi-line equations — is unmatched by any other widely available system.

Inline mathematics is enclosed in `$. . . $`:

```
Einstein's equation  $E = mc^2$  relates energy to mass.
The gradient is  $\nabla f = \sum_{i=1}^n \frac{\partial f}{\partial x_i} \hat{e}_i$ .
```

Displayed mathematics uses `\[. . . \]` for single equations and the `align` environment from `amsmath` for multi-line equations:

```
The fundamental theorem of calculus:
\[
\int_a^b f'(x) dx = f(b) - f(a)
\]

The align environment for multi-line equations:
\begin{align}
\nabla \cdot \mathbf{E} &= \frac{\rho}{\varepsilon_0} \\
\nabla \cdot \mathbf{B} &= 0 \\
\nabla \times \mathbf{E} &= -\frac{\partial \mathbf{B}}{\partial t} \\
\nabla \times \mathbf{B} &= \mu_0 \mathbf{J} + \mu_0 \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t}
\end{align}
```

The `&` character aligns the equals signs across all lines. The `\\` ends each line. Equations are numbered automatically; suppress numbering for a specific line with `\nonumber` or use the `align*` environment to suppress numbering throughout.

The **amsmath package** is essential for serious mathematical typesetting. It provides `align`, `gather`, `multline`, `split`, `cases`, improved fraction commands, and dozens of other enhancements over LaTeX's built-in mathematics. Load it in every mathematical document:

```
\usepackage{amsmath}
\usepackage{amssymb} % additional mathematical symbols
\usepackage{amsthm} % theorem environments
```

Theorem environments from `amsthm` are the standard for mathematical writing:

```
\usepackage{amsthm}

\newtheorem{theorem}[Theorem][section] % numbered within sections
\newtheorem{lemma}[theorem]{Lemma} % shares counter with theorem
\newtheorem{corollary}[theorem]{Corollary}

\theoremstyle{definition}
\newtheorem{definition}[theorem]{Definition}

\theoremstyle{remark}
\newtheorem*{remark}{Remark} % unnumbered
```

The three theorem styles are `plain` (bold label, italic text — for theorems, lemmas), `definition` (bold label, upright text — for definitions, examples), and `remark` (italic label, upright text — for remarks, notes). Choosing the right style for each environment is a typographic decision, not just an aesthetic one: the visual contrast between theorem and definition bodies signals the difference in their logical character.

Essential packages

Every serious LaTeX document uses a core set of packages. These are not optional enhancements — they correct genuine deficiencies in LaTeX's defaults.

geometry controls page dimensions and margins:

```
\usepackage{geometry}
\geometry{
  a4paper,
  top=30mm, bottom=25mm,
  left=30mm, right=25mm,
  bindingoffset=10mm % for two-sided printing
}
```

microtype enables microtypographic extensions: character protrusion (hanging punctuation into the margin) and font expansion (slight width variation to improve justification). The visual effect is subtle but the text quality improvement is real — justified paragraphs with fewer hyphens, more even spacing, and fewer overfull lines:

```
\usepackage{microtype}
```

With pdfLaTeX, full protrusion and expansion are available. With XeLaTeX, only protrusion is available. With LuaLaTeX, both are available and **microtype** has additional capabilities. Load **microtype** with no options and it applies sensible defaults for whichever engine you are using.

hyperref adds PDF navigation (clickable cross-references, table of contents, citations) and writes document metadata:

```
\usepackage{hyperref}
\hypersetup{
  pdftitle={The CLI Typographer},
  pdfauthor={A. N. Author},
  colorlinks=true,
```

```

linkcolor=NavyBlue,
urlcolor=Maroon,
citecolor=black
}

```

Load `hyperref` last in the preamble (or second-to-last if using `cleveref`, which must follow `hyperref`).

booktabs provides professional-quality table rules:

```

\usepackage{booktabs}

\begin{table}
\centering
\begin{tabular}{llc}
\toprule
Tool & Primary format & Released \\
\midrule
TeX & DVI & 1978 \\
LaTeX & DVI / PDF & 1985 \\
pdfTeX & PDF & 1996 \\
XeTeX & PDF & 2004 \\
LuaTeX & PDF & 2007 \\
\bottomrule
\end{tabular}
\caption{TeX engine timeline}
\label{tab:engines}
\end{table}

```

The `booktabs` rules — `\toprule`, `\midrule`, `\bottomrule` — have appropriate thickness and spacing. Do not use vertical rules in tables; the visual noise they add outweighs any organisational benefit.

enumitem replaces LaTeX's built-in list environments with fully customisable alternatives:

```

\usepackage{enumitem}

% Compact list with no extra spacing
\begin{itemize}[noitemsep]
\item First item

```

```

\item Second item
\end{itemize}

% Numbered list with custom label format
\begin{enumerate}[label=\textbf{\arabic*}, leftmargin=*]
\item Step one
\item Step two
\end{enumerate}

```

cleveref provides smart cross-references that automatically include the reference type:

```

\usepackage{cleveref}

% Instead of: see Table~\ref{tab:engines}
% Write:
See \cref{tab:engines}. % produces "see Table 1"
See \Cref{tab:engines}. % produces "see Table 1" (capitalised)
See \cref{sec:intro,sec:methods}. % multiple: "see Sections 1 and 3"

```

siunitx formats numbers and physical units consistently:

```

\usepackage{siunitx}

A document of \qty{350}{pages}.
A resolution of \qty{600}{dpi}.
A file size of \qty{2.4}{\mega\byte}.
A temperature of \qty{-17.3}{\celsius}.
Numbers with \num{1234567.89} formatted correctly.

```

siunitx handles the spacing between number and unit, the formatting of large numbers, and the typesetting of unit symbols correctly according to SI conventions.

Bibliography management

LaTeX has two bibliography systems: the older BibTeX with the natbib package, and the modern BibLaTeX with the biber backend. BibLaTeX/Biber is the current standard; use it for new documents unless you have a specific reason to use BibTeX.

BibTeX and **natbib** remain in widespread use because of their stability and because many publisher templates require them:

```
\usepackage{natbib}

% In the document body:
\citet{knuth1984}      % Knuth (1984)
\citep{knuth1984}     % (Knuth, 1984)
\citep[p.~42]{knuth1984} % (Knuth, 1984, p. 42)
\citealt{knuth1984}   % Knuth 1984 (no parentheses)

% At the end of the document:
\bibliographystyle{plainnat}
\bibliography{references}
```

Compile sequence for BibTeX: `pdflatex` → `bibtex` → `pdflatex` → `pdflatex`. The `latexmk` tool automates this sequence.

BibLaTeX with **Biber** provides a more powerful and flexible system, with better Unicode support, richer entry types, and more customisable citation and bibliography styles:

```
\usepackage[
  style=authoryear,      % or: numeric, chicago-notes, apa, ieee
  backend=biber,
  sortlocale=en_GB
]{biblatex}
\addbibresource{references.bib}

% In the document body:
\textcite{knuth1984}    % Knuth (1984)
\parencite{knuth1984}  % (Knuth, 1984)
\parencite[p.~42]{knuth1984}
\autocite{knuth1984}   % style-dependent: parens or footnote
```

```
% At the end of the document:
\printbibliography
```

Compile sequence for BibLaTeX/Biber: `xelatex` → `biber` → `xelatex` → `xelatex`. Again, `latexmk` handles this automatically with `$pdf_mode = 5` (XeLaTeX) and `$biber = 'biber %0 %S'` in `.latexmkrc`.

The `.bib` file format is the same for both systems — a collection of entries identifying each cited work by type and fields:

```
@book{knuth1984,
  author = {Knuth, Donald E.},
  title  = {The {\TeX}book},
  series = {Computers \& Typesetting},
  volume = {A},
  publisher = {Addison-Wesley},
  year   = {1984},
  isbn   = {0-201-13447-0}
}

@article{lampport1986,
  author = {Lampport, Leslie},
  title  = {Document Production: Visual or Logical?},
  journal = {Notices of the American Mathematical Society},
  year   = {1987},
  volume = {34},
  number = {5},
  pages  = {621--624}
}
```

Note `{\TeX}` to protect the `\TeX` command from case-folding by bibliography styles that format titles in title case. Protecting proper nouns and commands with braces is a BibTeX idiom.

Custom commands and environments

One of LaTeX's great strengths is the ability to define semantic markup — custom commands that encode meaning rather than appearance, so that the appearance can be changed globally by modifying the definition.

New commands with `\newcommand`:

```
% \newcommand{name}[num_args]{definition}

\newcommand{\cli}[1]{\texttt{#1}}
\newcommand{\pkg}[1]{\textsf{\small #1}}
\newcommand{\file}[1]{\texttt{#1}}
\newcommand{\term}[1]{\emph{#1}}

% Usage:
The \cli{fc-list} command is in the \pkg{fontconfig} package.
Open the \file{metadata.yaml} file.
The \term{baseline} is the invisible line on which text sits.
```

When you later decide that package names should be bold rather than small sans-serif, change `\pkg`'s definition in one place and every instance updates.

New environments with `\newenvironment`:

```
% \newenvironment{name}[num_args]{begin-code}{end-code}

\newenvironment{aside}{%
  \begin{quote}\small\itshape
}{%
  \end{quote}
}

\newenvironment{codeblock}[1][1]{%
  \begin{tcolorbox}[
    title=#1,
    fonttitle=\small\sffamily,
    colback=gray!5,
    colframe=gray!50
  ]
  \small\ttfamily
}{%
  \end{tcolorbox}
}
```

Custom counters enable numbered environments that integrate with LaTeX's cross-referencing system:

```

\newcounter{example}[section]
\renewcommand{\theexample}{\thesection.\arabic{example}}

\newenvironment{example}[1][1][%
  \refstepcounter{example}%
  \begin{tcolorbox}[title=Example \theexample\ifx&#1&\else: #1\fi]
}{%
  \end{tcolorbox}
}

% Usage:
\begin{example}[Font selection]
  The \cli{fc-match serif} command shows the default serif font.
\end{example}

```

`\refstepcounter` increments the counter and registers the current value for `\label/\ref`, so these examples can be cross-referenced like any other numbered element.

Custom document classes

A custom document class captures all the formatting decisions for a document type in a single reusable file. Once written, it can be applied to any document of that type without repeating the configuration.

A minimal class file:

```

% myarticle.cls
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{myarticle}[2024/01/01 Custom article class]

% Pass options to the base class
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}
\ProcessOptions\relax

% Load the base class
\LoadClass[11pt,a4paper]{article}

% Required packages
\RequirePackage[T1]{fontenc}

```

```

\RequirePackage{geometry}
\RequirePackage{microtype}
\RequirePackage{fancyhdr}
\RequirePackage{titlesec}
\RequirePackage{hyperref}
\RequirePackage{booktabs}

% Page geometry
\geometry{a4paper, top=30mm, bottom=25mm, left=30mm, right=25mm}

% Section heading style
\titleformat{\section}
  {\normalfont\large\bfseries}{\thesection}{1em}{}
\titleformat{\subsection}
  {\normalfont\normalsize\bfseries}{\thesubsection}{1em}{}

% Page style
\pagestyle{fancy}
\fancyhf{}
\fancyhead[L]{\small\textit{\leftmark}}
\fancyhead[R]{\small\thepage}
\renewcommand{\headrulewidth}{0.4pt}

% Paragraph spacing
\setlength{\parindent}{1.5em}
\setlength{\parskip}{0pt}

% Hyperref defaults
\hypersetup{colorlinks=true, linkcolor=NavyBlue, urlcolor=Maroon,
  ↪ citecolor=black}

\endinput

```

Place `myarticle.cls` in the same directory as the `.tex` file, or in a local TeX directory (typically `~/texmf/tex/latex/myarticle/`) where LaTeX will find it automatically. Then use it with:

```
\documentclass{myarticle}
```

This is the foundational pattern for Chapter 24's treatment of templates and style systems: encode decisions once, apply them everywhere.

The compilation workflow with latexmk

Direct `pdflatex` or `xelatex` invocations require you to manage multi-pass compilation manually — LaTeX needs multiple runs to resolve forward references, update the table of contents, and process bibliographies. `latexmk` automates this entirely.

A project-level `.latexmkrc` file configures `latexmk`'s behaviour:

```
# .latexmkrc

# Use XeLaTeX
$pdf_mode = 5;
$xelatex = 'xelatex -interaction=nonstopmode -synctex=1 %0 %S';

# Use Biber for bibliography
$biber = 'biber %0 %S';

# Clean up auxiliary files
$clean_ext = 'synctex.gz run.xml bbl bcf fdb_latexmk fls';

# Output directory (keeps source directory clean)
$out_dir = 'build';
```

With this configuration:

```
latexmk document.tex # build, running as many passes as needed
latexmk -pvc document.tex # preview-continuously: rebuild on save
latexmk -c # clean auxiliary files
latexmk -C # clean everything including the PDF
```

The `-pvc` flag is particularly useful during drafting: `latexmk` watches the source file and rebuilds automatically whenever you save. Combined with a PDF viewer that reloads automatically (most modern viewers do), this gives a near-live preview loop for LaTeX documents.

The `$out_dir = 'build'` setting keeps the source directory uncluttered — all auxiliary files and the final PDF are written to `build/`. This works seamlessly with the Makefile patterns from Chapter 10.

LaTeX is a large system and this chapter covers only its core. The document examples in Part IV — especially where publisher requirements or established classes still matter — demonstrate these techniques in context. Chapter 26 returns to LaTeX for microtypography, covering microtype's protrusion and expansion settings in detail and the techniques for eliminating widows, orphans, and bad line breaks in production-quality documents. The point is not that every new project should start in LaTeX; it is that when the ecosystem demands LaTeX, you should know what it is doing and how to use it well.

Typst — The Modern Alternative

For most of LaTeX's existence, it had no serious competitors for high-quality mathematical and scientific typesetting. Various attempts at simpler systems produced simpler output; systems with equivalent output quality required equivalent complexity. Typst, first released publicly in 2023, is the first system in forty years that genuinely challenges this situation. It produces LaTeX-quality output through a language that is dramatically cleaner, with compilation that is dramatically faster, and an error experience that is dramatically less painful.

This chapter covers Typst in depth: its syntax, its approach to typography, its scripting system, its package ecosystem, and where it currently falls short of LaTeX. The goal is to give you enough understanding to choose it when it is the right tool — which is increasingly often — and to know its limits when it is not.

What Typst is

Typst is a typesetting system with a built-in scripting language. The scripting language is not a macro preprocessor bolted onto a typesetting engine (which is essentially what LaTeX is): it is a coherent, functional programming language in which the entire document is a value. Text is a first-class value. Functions produce content. Content can be stored in variables, passed to functions, mapped over, filtered, and returned.

Two consequences follow from this architecture. First, the document language is consistent: there is one way to call a function, one way to define a variable, one way to loop, one way to apply conditional logic. LaTeX's patchwork of syntax conventions — `\command{arg}`, `\begin{env}...end{env}`, `\setcounter`, `\ifx...fi`, `\expandafter` —

is replaced by a small set of orthogonal constructs. Second, the compiler can understand the document structure as it compiles, enabling incremental compilation: change one paragraph, recompile only the affected pages. On a large document, Typst's compilation is measured in milliseconds rather than seconds.

The other major architectural difference from LaTeX is error reporting. LaTeX errors are famously opaque because TeX's macro expansion model makes it difficult to trace errors back to their source. Typst errors point to the specific line and character where the problem occurred, and the messages are written in plain English: `error: cannot divide by zero`, `error: expected string, found integer`.

Installation

Typst is a single binary with no external dependencies. Installation is a matter of downloading the binary for your platform and placing it in your PATH.

The recommended installation method uses the official install script:

```
curl -fsSL https://typst.community/typst-install/install.sh | sh
```

This installs the latest stable release to `~/.local/bin/typst`. Alternatively, install via your system's package manager where available, or via Cargo if you have a Rust toolchain:

```
cargo install typst-cli
```

Verify the installation:

```
typst --version
```

Typst requires no LaTeX installation, no font configuration, and no package manager setup. Fonts are discovered through the system's font infrastructure (fontconfig on Linux, Core Text on macOS), and packages from the Typst Universe registry are downloaded automatically on first use — no separate install step.

Basic syntax

A Typst document is a plain text file with the `.typ` extension. Three kinds of content can appear:

Markup mode is the default. Plain text is plain text. Headings begin with `=`. Lists begin with `-`. Formatting uses `*bold*`, `_italic_`, and ``code``. This will be immediately familiar from Markdown.

Code mode is entered with `#`. Everything following `#` until the end of the expression is interpreted as Typst code — a function call, a variable, a control structure, a set rule.

Math mode is entered with `$`. Single `$` produces inline math; double `$` on its own line produces a displayed equation.

```
= Introduction // heading (markup)

Body text is just text. *Bold* and _italic_ work as expected.
A footnote#footnote[Like this.] and some `inline code`.

#let greeting = "Hello" // variable definition (code)
#greeting, world. // variable interpolation

$E = m c^2$ // inline math
$ \int_a^b f'(x) \, dx = f(b) - f(a) $ // displayed math
```

The `#` prefix can be omitted for block-level code: a line that begins with `let`, `for`, `if`, or `show` is treated as code automatically. The `#` is required for inline code interpolation within a paragraph.

Set and show rules

The two most important language constructs in Typst are set and show rules. Together, they replace most of what LaTeX uses packages and class files for.

A set rule applies default parameters to a function for the rest of the document (or the current scope). This is how you configure page size, typography, paragraph spacing, heading numbering, and everything else:

```
// Page setup
#set page(
  paper: "a4",
  margin: (top: 30mm, bottom: 25mm, left: 30mm, right: 25mm),
  numbering: "1",
)

// Typography
#set text(
  font: "EB Garamond",
  size: 11pt,
  lang: "en",
)

// Paragraph settings
#set par(
  justify: true,
  leading: 0.65em,
  first-line-indent: 1.5em,
)

// Heading numbering
#set heading(numbering: "1.1")

// List appearance
#set list(indent: 1em)
```

Every Typst function that produces content accepts keyword arguments that control its appearance. set rules establish defaults for those arguments throughout the document or scope.

A show rule transforms how a specific kind of content is rendered. This is how you redefine the appearance of headings, code blocks, figures, bibliographies, or any other content type:

```
// Redefine level-1 headings
#show heading.where(level: 1): it => {
  v(2em, weak: true)
  block(
    below: 0.8em,
    text(size: 18pt, weight: "bold", fill: rgb("#1a1a2e"), it.body)
  )
}

// Add a background to all code blocks
#show raw.where(block: true): it => {
  block(
    fill: luma(245),
    inset: (x: 1em, y: 0.75em),
    radius: 4pt,
    width: 100%,
    text(size: 9.5pt, font: "JetBrains Mono", it)
  )
}

// Style blockquotes
#show quote: it => {
  pad(left: 1.5em, block(
    stroke: (left: 3pt + luma(180)),
    inset: (left: 1em),
    text(style: "italic", it.body)
  ))
}
```

Show rules can target content by type (heading, raw, figure), by attributes (`.where(level: 1)`), or by content matching ("important text"). They are significantly more powerful and composable than LaTeX's equivalent mechanisms.

Typography configuration

Typst's typography defaults are already good. The line-breaking algorithm is paragraph-level, like TeX's — it considers the whole paragraph when breaking lines, minimising total badness rather than breaking greedily. With appropriate font and layout settings, the output quality is close to LaTeX's.

Font selection names fonts as strings, matched against fontconfig (or the system's font discovery mechanism):

```
#set text(font: "EB Garamond", size: 11pt)
#set text(font: ("EB Garamond", "Linux Libertine", "serif"))
```

When a list of fonts is provided, Typst uses the first available one, falling back down the list. The final entry "serif" is a generic family name that resolves to the system's default serif font — a reliable ultimate fallback.

Specifying font paths for fonts not in the system font directories:

```
typst compile --font-path ./fonts document.typ
```

Multiple `--font-path` options are accepted. The Typst compiler searches these directories in addition to system font paths.

OpenType features are configured through text arguments:

```
#set text(
  font: "EB Garamond",
  features: ("onum", "liga", "kern"),
)
```

Or per-span for specific uses:

```
#text(features: ("smcp",))[Small capitals here]
```

Page headers and footers use the header and footer arguments to page, with the context keyword for accessing the current page state:

```
#set page(
  header: context {
    let page-num = counter(page).get().first()
    if page-num > 1 {
      grid(
        columns: (1fr, 1fr),
        align(left)[#smallcaps[The CLI Typographer]],
        align(right)[#page-num]
      )
      line(length: 100%, stroke: 0.4pt)
    }
  },
  footer: context {
    // No footer – page number is in the header
  }
)
```

The context keyword marks a region where the current document state (page number, current heading, counter values) can be queried. This is the Typst equivalent of LaTeX’s two-pass compilation — in Typst, context-sensitive content is resolved in a separate pass automatically.

Mathematics

Typst’s math mode uses a syntax designed for readability rather than direct compatibility with LaTeX. The key difference is that most mathematical symbols are written as words without backslashes, and the overall syntax is less cluttered.

Inline math uses $\$ \dots \$$ as in LaTeX:

```
Einstein’s equation  $E = m c^2$  relates energy to mass.
```

Displayed math uses $\$ \dots \$$ where the dollar signs are on their own lines:

The Basel problem:

```
$ sum_(n=1)^infinity 1/n^2 = pi^2/6 $
```

Key syntax differences from LaTeX math:

Greek letters and mathematical symbols are written as words without backslash: alpha, beta, Gamma, infinity, integral, sum, product. Common operators are written directly: \rightarrow for arrow, \leq for \leq , \neq for \neq .

Subscripts and superscripts use `_` and `^` as in LaTeX, but multi-character arguments use parentheses rather than braces: $x^{(n+1)}$, $\sum_{(i=1)}^n$. Single characters need no delimiter: x^2 , x_n .

Fractions are written with the division operator: a/b produces a fraction. For explicit fraction control, `frac(a, b)` is available.

Matrices and vectors use `mat(...)` and `vec(...)`:

```
$ mat(a, b; c, d) $           // 2x2 matrix (semicolon separates rows)
$ vec(x, y, z) $             // column vector
$ mat(delim: "[", a, b; c, d) $ // square brackets
```

Aligned equations use the `&` alignment marker and `\` for line breaks, as in LaTeX's `align` environment:

```
$ (x + y)^2 &= x^2 + 2 x y + y^2 \
(x - y)^2 &= x^2 - 2 x y + y^2 $
```

Theorem-like environments are not built in but are straightforwardly implemented with functions:

```
#let thmbox(name, body, color: luma(240)) = {
  block(
    fill: color,
    inset: 10pt,
    radius: 4pt,
    width: 100%,
    [*#name.* #body]
  )
}
```

```

}

#let theorem = thmbox.with("Theorem")
#let lemma   = thmbox.with("Lemma")
#let proof(body) = pad(left: 1em,
  [_Proof._ #body #h(1fr) $square$]
)

// Usage:
#theorem[There are infinitely many prime numbers.]
#proof[Suppose $p_1, \dots, p_n$ were all primes. Let
  $N = p_1 p_2 \dots c p_n + 1$. Then $N$ has no prime
  factors in our list – contradiction.]

```

For automatic theorem numbering, a counter is used:

```

#let thm-counter = counter("theorem")

#let theorem(body, name: none) = {
  thm-counter.step()
  let num = context thm-counter.display()
  let label = if name != none { " (" + name + ")" } else { "" }
  block(fill: luma(240), inset: 10pt, radius: 4pt,
    [*Theorem #num#label.* #body]
  )
}

```

Scripting and programmatic documents

Typst’s scripting capabilities are what set it apart from previous LaTeX alternatives. It is a proper programming language: variables, functions, conditionals, loops, arrays, dictionaries, and first-class functions.

Variables and functions:

```

#let pi-approx = 3.14159
#let square(x) = x * x

The area of a circle with radius 5 is approximately
#calc.round(pi-approx * square(5), digits: 2).

```

```
// Functions that return content
#let important(body) = {
  text(fill: rgb("#c0392b"), weight: "bold", body)
}
```

This is #important[critically important].

Loops and conditionals:

```
// Generate a multiplication table
#table(
  columns: 5,
  ..for i in range(1, 6) {
    for j in range(1, 6) {
      (str(i * j),)
    }
  }
)

// Conditional content
#let draft = true
#if draft {
  text(fill: red)[DRAFT – DO NOT DISTRIBUTE]
}
```

Reading external data — useful for generated reports and documents that must reflect current data:

```
// Read a CSV file and render it as a table
#let data = csv("data.csv")
#table(
  columns: data.first().len(),
  ..data.flatten()
)
```

Building a letter template demonstrates the scripting approach to document components:

```
#let letter(  
  sender: "",  
  recipient: "",  
  date: none,  
  subject: none,  
  body  
) = {  
  set page(paper: "a4", margin: (top: 35mm, bottom: 30mm,  
                                left: 30mm, right: 25mm))  
  set text(font: "EB Garamond", size: 11pt)  
  set par(justify: true)  
  
  // Sender address (right-aligned)  
  align(right, text(size: 10pt, sender))  
  v(1em)  
  
  // Date  
  if date != none { align(right, date); v(1em) }  
  
  // Recipient address  
  text(recipient)  
  v(2em)  
  
  // Subject line  
  if subject != none {  
    text(weight: "bold", "Subject: " + subject)  
    v(1em)  
  }  
  
  // Body  
  body  
  
  v(2em)  
  text("Yours sincerely,")  
  v(3em)  
  // Signature space  
  text(sender.split("\n").first())  
}
```

Templates and reuse

A Typst template is a `.typ` file that defines functions and set/show rules, imported into document files with `#import`. This is the mechanism that replaces LaTeX document classes.

A template file `article-template.typ`:

```
#let article(
  title: none,
  authors: (),
  date: none,
  abstract: none,
  body
) = {
  set document(title: title, author: authors.map(a => a.name))

  set page(paper: "a4", margin: (x: 30mm, y: 30mm),
    numbering: "1")
  set text(font: "New Computer Modern", size: 11pt, lang: "en")
  set par(justify: true, leading: 0.65em,
    first-line-indent: 1.5em)
  set heading(numbering: "1.1")

  // Title block
  align(center)[
    #block(text(size: 18pt, weight: "bold", title))
    #v(0.5em)
    #authors.map(a => block[#a.name, _#a.affiliation._]).join()
    #if date != none { block(text(size: 10pt, date)) }
  ]

  if abstract != none {
    v(1em)
    block(inset: (x: 2em))[*Abstract.* #abstract]
    v(1em)
  }

  body
}
```

The document that uses it:

```
#import "article-template.typ": article

#show: article.with(
  title: "On CLI Typesetting",
  authors: (
    (name: "A. N. Author", affiliation: "University of Example"),
  ),
  date: "March 2024",
  abstract: [A practical guide to document production tools.],
)

= Introduction

Body text begins here.
```

The `#show: template.with(args)` pattern applies the template's formatting to the entire document. This is standard Typst idiom for reusable templates.

Published templates are distributed as Typst Universe packages, installed automatically on first import:

```
#import "@preview/charged-ieee:0.1.2": ieee
#show: ieee.with(
  title: [Paper Title],
  authors: ((name: "Author Name", ...)),
)
```

The `@preview/` prefix identifies packages from the official Typst Universe registry, versioned with semantic version numbers. Packages are cached locally after first download and never fetched again for the same version, making builds reproducible.

Bibliography and citations

Typst handles bibliography natively using BibTeX `.bib` files:

```
#bibliography("references.bib")
```

Citations use the @key syntax:

```
As described in @knuth1984, the line-breaking algorithm...
The standard reference is @bringhurst2004[p. 17].
```

Bibliography styles are selected from Typst's built-in set or from CSL files:

```
// Built-in styles
#bibliography("references.bib", style: "ieee")
#bibliography("references.bib", style: "apa")
#bibliography("references.bib", style: "chicago-author-date")

// CSL style
#bibliography("references.bib", style: "my-style.csl")
```

The default style is alphabetical (author-year sorted); common alternatives include "ieee", "apa", "mla", "chicago-author-date", and "chicago-notes".

The package ecosystem

Typst packages live on **Typst Universe** (typst.app/universe) and are imported with `@preview/package-name:version`. As of early 2025, the ecosystem is growing rapidly. Packages worth knowing:

@preview/codly provides code listing environments with line numbers, highlights, and zebra striping — a richer alternative to Typst's built-in raw blocks.

@preview/cetz is a drawing library for technical illustrations, analogous to TikZ. It uses a Typst-native API for paths, shapes, and connectors.

@preview/fletcher extends `cetz` specifically for diagrams: flowcharts, commutative diagrams, state machines. The API is cleaner than `TikZ` for this class of diagram.

@preview/showybox provides styled boxes for callouts, warnings, examples, and other highlighted content.

@preview/unify handles physical quantities and units, analogous to LaTeX's `siunitx`.

@preview/tablex extends Typst's table system with merged cells, custom strokes, and other features not available in the built-in `table` function.

The ecosystem is smaller than CTAN, and some LaTeX packages have no Typst equivalent yet. The most notable gaps are journal-specific templates (most academic publishers have LaTeX templates only), certain specialised mathematical environments, and the more niche typographic packages. For standard academic and technical documents, these gaps rarely matter.

Current limitations

Typst is a young system and some limitations are worth knowing before committing to it for a long project.

No CTAN package compatibility. LaTeX packages do not work in Typst. If your document depends on a specific LaTeX package with no Typst equivalent, you cannot use Typst for that document.

Publisher submission requirements. Most academic publishers and conference proceedings accept only LaTeX (or sometimes Word). If you need to submit source files to a publisher, Typst will not be accepted unless the publisher specifically supports it. You can generate a PDF from Typst for preprints and personal use, but the submission workflow may require LaTeX.

BibLaTeX-style bibliography features. Typst's bibliography system is BibTeX-level: it handles standard citation types well, but the advanced

bibliography customisation available through BibLaTeX (custom entry types, annotation fields, complex inheritance) is not yet available.

Multi-column layouts are possible in Typst but more limited than in LaTeX. Documents with complex flowing multi-column text (two-column academic papers with figures that span both columns) require more work in Typst than in LaTeX.

Change tracking and revision marks. LaTeX's changes package and related tools have no Typst equivalent; collaborative editing with tracked changes is not natively supported.

These limitations are real but bounded. For the majority of technical documents — papers, reports, books, theses, technical documentation — Typst is fully capable and offers a substantially better authoring experience than LaTeX. The decision to use Typst or LaTeX is increasingly a matter of specific requirements rather than general capability.

Compilation workflow

Typst's single-binary architecture means the compilation workflow is simple:

```
# Compile once
typst compile document.typ

# Compile with custom output path
typst compile document.typ build/output.pdf

# Compile with custom font paths
typst compile --font-path ./fonts document.typ

# Watch mode: recompile on every save
typst watch document.typ

# Watch mode with PDF viewer auto-open
typst watch --open document.typ
```

The `watch` command is the primary development workflow: run it once, save the file, and the PDF updates within a fraction of a second. For large documents, the incremental compilation means even complex multi-page documents rebuild nearly instantly after small changes.

Integration with Make follows the same pattern as LaTeX:

```
SOURCES := $(wildcard chapters/*.typ) template.typ
BUILD   := build

$(BUILD)/book.pdf: $(SOURCES) | $(BUILD)
    typst compile --font-path fonts/ chapters/main.typ $@

$(BUILD):
    mkdir -p $@
```

For multi-file projects, Typst handles imports automatically — there is no need to pass all files on the command line. A main entry file imports the others:

```
// main.typ
#import "template.typ": article
#show: article.with(...)

// Include chapters
#include "chapters/01-history.typ"
#include "chapters/02-fundamentals.typ"
#include "chapters/03-digital.typ"
```

Changes to any included file trigger a recompile of the output, handled transparently by `typst watch`.

Typst and LaTeX will coexist for years, probably for decades. LaTeX has too much accumulated infrastructure — too many journal templates, too many existing documents, too deep an ecosystem — to be displaced quickly. But for new projects where the choice is open, Typst is now a

serious option wherever LaTeX would otherwise be the default. The chapters that follow — Quarto for computational documents, Emacs and Org Mode for Emacs users — cover the remaining tools in Part III's toolbox before Part IV puts all of them to work on real document examples.

Quarto

There is a category of document that Pandoc and LaTeX do not handle well: the document where the content is produced by computation. An analysis in which the charts, tables, and summary statistics are generated by code — Python, R, Julia — and where the narrative and the code are the same artifact, not two separate files that must be kept in sync. When the data changes, or the analysis is revised, the document must be regenerated with the new results embedded correctly. This is the reproducible research problem, and it is the problem Quarto was built to solve.

Quarto is Posit's open-source scientific and technical publishing system, released in 2022 as the successor to R Markdown. It extends Pandoc Markdown with executable code blocks in Python, R, Julia, and Observable JavaScript, and it wraps the whole system in a polished publishing framework with support for articles, books, websites, presentations, and interactive dashboards. If Pandoc is the universal converter, Quarto is the publishing system that turns Markdown plus computation into a practical multi-format workflow. Its PDF output can ride on Typst or LaTeX; the important point is that the source remains Markdown and the workflow remains reproducible.

This chapter covers Quarto from the CLI author's perspective: its document format, its execution model, its project types, its customisation system, and the patterns that make reproducible documents practical.

Installation and setup

Quarto is a standalone tool that installs separately from R or Python. The installation includes Pandoc (a specific version bundled with Quarto) and does not depend on a system Pandoc installation.

```
# Download and install on Linux
curl -LO https://quarto.org/download/latest/quarto-linux-amd64.deb
sudo dpkg -i quarto-linux-amd64.deb

# Verify installation
quarto check
```

`quarto check` is essential after installation — it reports which execution engines are available and whether their dependencies are satisfied:

```
[ ] Checking versions of quarto binary dependencies...
    Pandoc version 3.1.9: OK
[ ] Checking versions of quarto dependencies.....OK
[ ] Checking Quarto installation.....OK

[ ] Checking Python 3 installation....Python 3.12.3
    Jupyter: 5.7.2
    Kernels: python3, ir

[ ] Checking R installation.....R 4.3.2
    knitr: 1.45

[ ] Checking tools.....OK
```

The output reveals which engines are available. For Python execution, Quarto uses Jupyter; for R, it uses knitr. Both can coexist in the same installation and even in the same document.

The QMD format

Quarto documents use the `.qmd` file extension (Quarto Markdown). They are Pandoc Markdown with two additions: a richer YAML front matter system that drives format selection and rendering options, and executable code blocks.

The YAML front matter controls every aspect of output: which formats to produce, what settings to apply to each, document metadata, bibliography, and execution behaviour. Unlike Pandoc's single output path per invocation, a QMD file can declare multiple output formats and Quarto will render all of them:

```
---
title: "Analysis of Typeface Legibility"
author:
  - name: "A. N. Author"
    affiliation: "University of Example"
    email: author@example.edu
date: "2024-03-15"
format:
  html:
    toc: true
    toc-depth: 3
    number-sections: true
    code-fold: true
  pdf:
    geometry: margin=25mm
    mainfont: "EB Garamond"
    pdf-engine: typst
bibliography: references.bib
---
```

Running `quarto render document.qmd` renders all declared formats. Running `quarto render document.qmd --to html` renders only HTML. The YAML metadata is automatically translated to the appropriate Pandoc options and template variables for each format.

Executable code blocks

A code block becomes executable by giving its language in curly braces: ````{python}` instead of ````python`. When Quarto encounters an executable block, it passes the code to the appropriate execution engine, captures the output, and embeds both the code and output in the rendered document.

```
#| label: fig-typeface-comparison
#| fig-cap: "Reading speed by typeface at three sizes"
#| echo: false
#| warning: false

import matplotlib.pyplot as plt
import numpy as np

typefaces = ["Garamond", "Palatino", "Times New Roman"]
sizes_pt = [9, 11, 14]
# Hypothetical reading speeds (wpm)
speeds = np.array([
    [220, 255, 268],
    [225, 258, 271],
    [218, 250, 265],
])

fig, ax = plt.subplots(figsize=(7, 4))
x = np.arange(len(typefaces))
width = 0.25
for i, size in enumerate(sizes_pt):
    ax.bar(x + i*width, speeds[:, i], width, label=f"{size}pt")
ax.set_xticks(x + width)
ax.set_xticklabels(typefaces)
ax.set_ylabel("Words per minute")
ax.legend(title="Size")
plt.tight_layout()
plt.show()
```

The lines beginning with `#|` are *chunk options* — YAML-formatted settings for the specific block. The most important chunk options:

Visibility: - `echo: true/false` — show or hide the source code
 - `output: true/false` — show or hide the output - `include:`

true/false — include or exclude the entire block (code and output) - warning: false — suppress warning messages - error: false — suppress errors (document fails if an error occurs)

Figures: - label: fig-name — assigns a cross-reference label (must begin fig- for figures) - fig-cap: "Caption text" — figure caption - fig-width: 6 — figure width in inches - fig-height: 4 — figure height in inches - fig-format: svg — output format (png, svg, pdf) - fig-dpi: 300 — resolution for raster formats

Tables: - label: tbl-name — table cross-reference label (must begin tbl-) - tbl-cap: "Caption" — table caption

Execution: - eval: false — show code but do not execute it - cache: true — cache results to avoid re-running expensive computations - dependson: "other-chunk" — invalidate cache if another chunk changes

Cross-references to figures and tables use the same @label syntax as citations:

```
As shown in @fig-typeface-comparison, reading speed increases with
font size across all three typefaces (see also @tbl-summary-stats).
```

This produces “As shown in Figure 1” in the output, with a clickable hyperlink in HTML and a correct reference in PDF. The system handles numbering automatically, even across output formats where figures are numbered differently.

Inline code and computed values

Beyond code blocks, individual values computed in code can be interpolated into prose. In Python documents:

```
The dataset contains {python} len(df) observations collected
between {python} df.date.min().strftime('%B %Y') and
{python} df.date.max().strftime('%B %Y').
```

In R documents, the syntax uses backtick-r:

```
The mean reading speed was `r round(mean(speeds), 1)` words per minute  
(SD = `r round(sd(speeds), 1)`).
```

This pattern — embedding computed values directly in prose — is the core of the reproducible document workflow. When the data changes and the document is re-rendered, every computed value updates automatically. The prose never drifts from the analysis.

Project types

A standalone `.qmd` file is suitable for single documents. For larger work, Quarto *projects* provide structure: a `_quarto.yml` configuration file defines the project type, shared settings, and the collection of documents that constitute the project.

Books

A Quarto book renders a collection of `.qmd` files as a multi-format book with automatic chapter numbering, cross-references that work across chapters, and a navigation interface in the HTML output.

The `_quarto.yml` for a book:

```
project:  
  type: book  
  output-dir: _book  
  
book:  
  title: "The CLI Typographer"  
  author: "A. N. Author"  
  date: "2024"  
  cover-image: assets/cover.jpg  
  
chapters:  
  - index.qmd
```

```

- part: "Foundations"
  chapters:
    - chapters/01-history.qmd
    - chapters/02-fundamentals.qmd
- part: "The Workflow"
  chapters:
    - chapters/03-markdown.qmd
    - chapters/04-pandoc.qmd
- references.qmd

bibliography: references.bib

format:
  html:
    theme: [cosmo, custom.scss]
    number-sections: true
    toc-depth: 3
  pdf:
    geometry: "margin=25mm, bindingoffset=10mm"
    mainfont: "EB Garamond"
    pdf-engine: typst
  epub:
    cover-image: assets/cover.jpg

```

The `part:` key introduces a named part containing sub-chapters — an optional layer of hierarchy above chapters. The `index.qmd` file is the book’s front matter or preface; `references.qmd` is a conventional name for the bibliography page, which Quarto appends automatically.

Running `quarto render` in the project directory builds all declared formats. The HTML output is a complete navigable website with a sidebar showing the book’s structure; the PDF is a typeset book; the EPUB is a properly structured ebook.

Websites

The `website` project type produces a multi-page HTML site from a collection of QMD files:

```
project:
  type: website

website:
  title: "CLI Typography Documentation"
  navbar:
    left:
      - href: index.qmd
        text: Home
      - href: guide/installation.qmd
        text: Guide
      - href: reference.qmd
        text: Reference
  sidebar:
    - title: "Guide"
      contents:
        - guide/installation.qmd
        - guide/first-document.qmd
        - guide/advanced.qmd

format:
  html:
    theme: flatly
    toc: true
    code-copy: true
```

Presentations

Quarto produces presentations in two formats from the same source: HTML slides using Reveal.js and PDF slides using Beamer. In line with the rest of this book, the HTML presentation is the primary path and the PDF deck is the fallback for venues that require it.

```
---
title: "CLI Typography"
format:
  revealjs:
    theme: default
    slide-number: true
    transition: fade
  beamer:
    theme: metropolis
```

```
aspectratio: 169
```

Slide breaks are created with `##` headings (level 2). The presentation body uses the same Markdown constructs as any Quarto document, plus a few presentation-specific features:

```
## The problem {.smaller}

::: {.incremental}
- Documents built with GUI tools encourage visual fiddling
- Reproducibility requires scripts, not mouse clicks
- Multiple output formats need automation
:::

## A demonstration {background-color="#2c3e50"}

```{python}
#| echo: true
import pandas
Live code in a slide
```

## Two-column layout

::::: {.columns}
::: {.column width="60%"}
Main content in the larger column.
:::
::: {.column width="40%"}
Supporting content in the narrower column.
:::
:::::
```

The `.incremental` class on a list makes items appear one at a time in the HTML presentation. The `{background-color}` attribute sets the slide's background. The `{.smaller}` class reduces font size on a specific slide.

Callouts and special blocks

Quarto provides five types of callout blocks for emphasising content — note, tip, warning, caution, and important — which render appropriately across formats:

```

::: {.callout-note}
## Note
This is a note callout. It uses blue styling in HTML and a
bordered box in PDF.
:::

::: {.callout-warning}
Always embed fonts before sending a PDF to a printer.
:::

::: {.callout-tip}
## Performance tip
Use `--freeze` in your `_quarto.yml` to cache computation
results across sessions.
:::

```

The callout type determines the styling: note is blue, tip is green, warning is yellow, caution is orange, and important is red. In PDF output, they render as coloured boxes; in HTML, as styled alert-style blocks; in Word output, as formatted sidebars.

Panel tabsets produce content in multiple tabs in HTML output and as sequential sections in PDF:

```

::: {.panel-tabset}
### Python
```python
result = sum(range(100))
print(result)
```

### R
```r
result <- sum(seq(0, 99))
print(result)

```

```
...

Shell
```sh  
python3 -c "print(sum(range(100)))"  
...  
:::
```

Themes and customisation

Quarto's HTML output uses Bootstrap 5 themes. The theme key in format: `html` accepts either a built-in theme name or a list combining a base theme with a custom SCSS file:

```
format:  
  html:  
    theme: [cosmo, custom.scss]
```

The custom SCSS file uses Quarto's theme variable system:

```
/*-- scss:defaults --*/  
  
// Typography  
$font-family-base: "EB Garamond", Georgia, serif;  
$font-size-base: 1.1rem;  
$line-height-base: 1.65;  
  
// Headings  
$headings-font-family: "Fira Sans", system-ui, sans-serif;  
$headings-font-weight: 500;  
  
// Colors  
$primary: #1a4e8c;  
$body-bg: #ffffff;  
$body-color: #1c1c1c;  
  
/*-- scss:rules --*/  
  
// Additional CSS rules
```

```

.callout {
  border-radius: 4px;
}

code {
  font-family: "JetBrains Mono", monospace;
  font-size: 0.875em;
}

pre.sourceCode {
  background-color: #f8f8f8;
  border-left: 3px solid #ddd;
}

```

The `/*-- scss:defaults --*/` section sets Bootstrap variable values. The `/*-- scss:rules --*/` section adds arbitrary CSS rules that complement rather than override the theme.

For PDF output, the equivalent customisation goes through the selected PDF backend. When the PDF path is Typst, keep those rules in the Typst layer. When compatibility requirements force a LaTeX backend, the `include-in-header` key injects raw LaTeX into the preamble:

```

format:
  pdf:
    documentclass: scrartcl
    mainfont: "EB Garamond"
    pdf-engine: xelatex
    include-in-header:
      text: |
        \usepackage{microtype}
        \usepackage{fancyhdr}
        \pagestyle{fancy}
        \fancyhead[L]{\textit{\leftmark}}
        \fancyhead[R]{\thepage}

```

Extensions

Quarto's extension system provides pre-packaged formats, filters, and shortcodes. Extensions are installed per-project from GitHub repositories:

```
# Install an extension
quarto add quarto-ext/lightbox           # clickable lightbox for images
quarto add quarto-ext/fontawesome       # Font Awesome icons
quarto add quarto-journals/jss          # J. Statistical Software format
quarto add quarto-journals/elsevier     # Elsevier journal template
quarto add quarto-journals/acm          # ACM proceedings format
```

Journal extensions are the most important category for academic authors. They provide format-specific templates, document class files, and submission-ready output without requiring LaTeX expertise:

```
---
title: "A Reproducible Analysis"
format:
  jss-pdf:
    keep-tex: true
  jss-html: default
---
```

After installing the JSS extension, `jss-pdf` and `jss-html` become available as output formats. The extension handles all the template details; the author writes standard Quarto Markdown.

Code execution and reproducibility

Quarto's approach to reproducibility centres on two mechanisms: caching and freezing.

Caching (`cache: true` in a code block or globally in `_quarto.yml`) stores execution results on disk. On subsequent renders, cached blocks are not re-executed unless their code changes or their dependencies change. The cache lives in a `_cache/` directory alongside the document.

```
# In _quarto.yml: cache all blocks by default
execute:
  cache: true
```

The limitation of caching is that it is session-local: the cache is not portable across machines and is not committed to version control. Collaborators rebuilding the document from scratch will re-execute everything.

Freezing (`freeze: auto` or `freeze: true`) commits execution results to the `_freeze/` directory, which is tracked in version control. When a collaborator without the required Python or R environment renders the document, frozen results are used rather than re-executing:

```
# In _quarto.yml
execute:
  freeze: auto # re-execute only when source changes
```

`freeze: auto` is the recommended setting for collaborative projects. It re-runs code when the source changes but uses frozen results otherwise. `freeze: true` never re-runs, which is appropriate for finalized data that should not change.

The `_freeze/` directory should be committed to version control alongside the source. This means contributors who do not have the computational environment installed can still build the full document:

```
# Full render (re-runs all code)
quarto render --execute

# Render using frozen results only
quarto render # uses _freeze/ where available
```

Comparison with R Markdown

Quarto supersedes R Markdown, though R Markdown remains functional and widely used. The key differences matter most for authors choosing between them for new projects.

Quarto is multi-language by design: the same syntax and tooling works for Python, R, Julia, and Observable without requiring separate packages (reticulate, JuliaCall, etc.). An R Markdown document that uses Python via reticulate requires a working R environment even though the Python code could run without R; the equivalent Quarto document uses a Jupyter kernel and requires only Python.

Quarto's chunk option syntax uses the YAML `#|` style rather than R Markdown's comma-separated `{r, options}` style. Both syntaxes work in Quarto for backward compatibility, but `#|` is preferred because it is consistent, readable, and does not require knowledge of R's syntax rules for quoting.

Quarto's cross-reference system is built-in and consistent across output formats, replacing the format-specific systems of bookdown, of f i cedown, and other R Markdown extensions. One syntax works for HTML, PDF, EPUB, and Word.

For existing R Markdown projects, migration to Quarto is straight-forward: rename `.Rmd` to `.qmd`, update the YAML header to Quarto's format, and run `quarto render`. Most documents require minimal changes.

A complete project workflow

The typical Quarto workflow for a computational document project:

```
# Create a book project
quarto create-project mybook --type book
cd mybook

# Structure created:
# _quarto.yml
# index.qmd
# intro.qmd
# summary.qmd
# references.bib

# Preview during development (live reload)
```

```
quarto preview

# Render all formats
quarto render

# Render specific format
quarto render --to pdf

# Render with fresh execution (ignore freeze)
quarto render --execute

# Publish to GitHub Pages
quarto publish gh-pages

# Publish to Quarto Pub (free hosting)
quarto publish quarto-pub
```

The `quarto preview` command is equivalent to `make watch` from Chapter 10 but integrated: it starts a local web server, opens the document in a browser, and rebuilds on every save. For HTML output, the rebuild is fast; for PDF output, the dedicated PDF backend runs on every save. In practice, use `quarto preview` for HTML drafting and `render` to PDF for production builds.

Quarto's position in the tool landscape is clear: if your document involves executable code that generates its content and you want HTML, PDF, and often EPUB or website output from the same source, Quarto is the right tool. Its handling of reproducibility, its multi-format output, and its polished publishing infrastructure are together the best available solution for computational documents. For documents with no computational content, its advantages over plain Pandoc are smaller — the additional YAML configuration and the Quarto layer between Markdown and Pandoc add complexity without adding capability. Chapter 11's decision framework applies: choose the tool that matches the document.

The next chapter examines Emacs and Org Mode — the oldest and in some ways most powerful system in this part of the book, with a learning curve to match.

Emacs and Org Mode

Emacs is the oldest continuously developed software environment still in active use. Created by Richard Stallman in 1976, it has accumulated decades of packages, conventions, and users whose workflows are built around it so completely that they author, manage tasks, read email, browse the web, and write code without leaving its interface. For document production, the relevant part of this ecosystem is Org Mode: a major mode that turns Emacs into one of the most capable document authoring environments available, with native support for outline-based writing, executable code blocks, table computation, and export to a wide range of output formats.

This chapter is addressed to two audiences. The first is the Emacs user who wants to understand Org Mode's document production capabilities more fully. The second is the non-Emacs user who keeps hearing about Org Mode and wants to know whether it is worth learning Emacs to use it. For the first audience, this chapter covers Org's document format, export system, Babel for literate programming, and AUCTeX for LaTeX authoring in depth. For the second: the answer is in the opening section, and it is honest.

Should you learn Emacs for Org Mode?

The honest answer is: probably not, unless you have other reasons to be in Emacs.

Org Mode is genuinely excellent. Its outline-based editing is elegant. Its table editing — with built-in spreadsheet functions — is remarkable. Babel, Org's code execution system, predates Quarto by over a decade and remains competitive with it for certain workflows. The export system

produces high-quality HTML, PDF, EPUB, and ODT from a single source file with fewer dependencies than a Pandoc-based pipeline.

The caveat is the Emacs prerequisite. Emacs has a learning curve unlike almost any other piece of software in current use. Its default keybindings predate the GUI conventions that have governed software interfaces since the 1980s. Its configuration is written in Emacs Lisp, a dialect of Lisp that is not casually approachable. Its documentation, while extensive, assumes a level of familiarity with the environment that newcomers cannot have. Getting Emacs to behave like a modern text editor for general use — sensible defaults, mouse support, sane keybindings — requires hours of configuration work before you can be productive.

None of these are criticisms. They are features for the people who have made the investment. But for someone who currently works in VS Code, Vim, or any other text editor, adopting Emacs for the sake of Org Mode means acquiring an entire new environment to gain capabilities that Pandoc, Quarto, Typst, and LaTeX already cover in other ways. The tradeoff is rarely worth it for document production alone.

If you are already an Emacs user, read on: Org Mode deserves your serious attention.

The Org format

Org is a plain text format built around the outline. Documents are structured as a hierarchy of headings, each introduced by one or more asterisks:

```
* First-level heading
** Second-level heading
*** Third-level heading
**** Fourth-level heading
```

Between headings is body text, which can contain any of Org's inline markup, links, tables, lists, and code blocks. The entire document is a tree, and Emacs can collapse or expand any subtree with a single

keystroke — showing only headings for navigation, or expanding a single section for focused writing, or displaying the whole document.

Inline markup uses symmetric delimiters:

<code>*bold*</code>	<code>/italic/</code>	<code>_underline_</code>
<code>=verbatim=</code>	<code>~code~</code>	<code>+strikethrough+</code>

The distinction between `=verbatim=` and `~code~` is meaningful in export: `=verbatim=` produces `<code>` in HTML and `\verb|...|` in LaTeX; `~code~` produces `<code>` in HTML and the same in LaTeX but is intended for code identifiers rather than literal text. In practice, both are often used interchangeably.

Links follow the double-bracket pattern:

<code>[[https://orgmode.org][Org Mode website]]</code>	← external link with ↪ description
<code>[[https://orgmode.org]]</code>	← bare URL
<code>[[./figures/diagram.png]]</code>	← inline image (no ↪ description)
<code>[[#custom-id]]</code>	← link to heading by ↪ CUSTOM_ID
<code>[[file:other-document.org::*Heading name]]</code>	← link to heading in ↪ another file

Footnotes have two syntaxes:

Inline footnote[fn:: The content goes here inline.]

Named footnote[fn:name] where the definition appears elsewhere.

[fn:name] The footnote content, which can span multiple lines and include markup.

Tables are Org's standout feature among plain-text formats. They are created and formatted automatically as you type, and they support spreadsheet-style formulas:

Tool	Year	Primary output
TeX	1978	DVI
LaTeX	1985	PDF
Typst	2023	PDF

Pressing TAB inside a table moves to the next cell, creating rows and re-aligning columns automatically. The `|-` prefix on a line creates a horizontal rule. Formulas are defined with `#+TBLFM:` directives below the table.

Structural keywords control the document’s metadata and export behaviour:

```
#+TITLE: The CLI Typographer
#+SUBTITLE: Typography and Typesetting from the Command Line
#+AUTHOR: A. N. Author
#+EMAIL: author@example.edu
#+DATE: 2024-03-15
#+LANGUAGE: en
#+OPTIONS: toc:2 num:t H:3 ^:nil
#+DESCRIPTION: A guide to CLI typesetting tools
```

The `#+OPTIONS:` keyword is a space-separated list of key:value pairs that control export behaviour. The most frequently needed options:

- `toc:2` — include table of contents to depth 2 (or `toc:nil` to suppress)
- `num:t` — number headings (or `num:nil` to suppress)
- `H:3` — maximum heading level (deeper headings become lists in some backends)
- `^:nil` — disable `_` and `^` as subscript/superscript markers (essential for text with underscores)
- `timestamp:nil` — suppress the “exported on” timestamp
- `author:nil` — suppress the author line

The export system

Org's export system is built from *export backends*, each of which is a module (ox-html, ox-latex, ox-odt, etc.) that knows how to render Org elements to a specific format. The interactive export dispatcher, invoked with C-c C-e, presents a menu of all available backends and their sub-commands. The most-used commands:

- C-c C-e l p — export to PDF via LaTeX
- C-c C-e l l — export to LaTeX source file
- C-c C-e h h — export to HTML file
- C-c C-e h o — export to HTML and open in browser
- C-c C-e o o — export to ODT

For command-line use without an interactive Emacs session, `--batch` mode invokes export functions directly:

```
# Export to HTML
emacs --batch \
  --eval "(require 'org)" \
  --visit "document.org" \
  --funcall org-html-export-to-html

# Export to PDF via LaTeX
emacs --batch \
  --eval "(require 'org)" \
  --eval "(setq org-latex-pdf-process '(\"latexmk -xelatex -quiet %f\"))"
↵ \
  --visit "document.org" \
  --funcall org-latex-export-to-pdf
```

The `--batch` flag runs Emacs without a display and exits after the commands complete. This makes Org export scriptable and usable in CI/CD pipelines — the same Make + Git + CI patterns from Chapter 10 apply directly, with `emacs --batch` replacing `pandoc`.

Loading a full user configuration in batch mode:

```
emacs --batch \
  --load "~/.emacs.d/init.el" \
  --visit "document.org" \
  --funcall org-latex-export-to-pdf
```

This applies all your custom LaTeX classes, package settings, and export options. The startup time is longer (it loads your full configuration), but the export uses exactly the settings you have developed interactively.

HTML export

The HTML backend (`ox-html`) produces HTML₅ by default. It includes a built-in CSS stylesheet, which can be suppressed and replaced with a custom one:

```
#+OPTIONS: html-style:nil
#+HTML_DOCTYPE: html5
#+HTML_HEAD: <link rel="stylesheet" href="style.css"/>
#+HTML_HEAD: <meta name="viewport" content="width=device-width">
```

The `#+HTML_HEAD:` keyword injects arbitrary content into the `<head>` element — the correct place for stylesheets, web fonts, and meta tags. Multiple `#+HTML_HEAD:` lines are all included.

Per-section HTML customisation uses property drawers:

```
*** Featured section
:PROPERTIES:
:HTML_CONTAINER_CLASS: featured
:END:
```

This section gets an additional CSS class on its container element.

Raw HTML for elements that Org cannot express:

```
#+BEGIN_EXPORT html
<figure class="wide">
  
  <figcaption>A diagram with a custom container class.</figcaption>
</figure>
#+END_EXPORT
```

Export-specific blocks are ignored by other backends — LaTeX export ignores `#+BEGIN_EXPORT html` blocks and vice versa. This allows format-specific content in a single source file.

LaTeX and PDF export

The LaTeX backend (`ox-latex`) translates Org structure to LaTeX commands and optionally runs the LaTeX compiler to produce a PDF. Document classes, packages, and preamble additions are configured through `#+LATEX_CLASS:`, `#+LATEX_CLASS_OPTIONS:`, and `#+LATEX_HEADER:` keywords:

```
#+LATEX_CLASS: scrartcl
#+LATEX_CLASS_OPTIONS: [11pt,a4paper]
#+LATEX_HEADER: \usepackage{fontspec}
#+LATEX_HEADER: \setmainfont{EB Garamond}
#+LATEX_HEADER: \usepackage{microtype}
#+LATEX_HEADER: \usepackage{geometry}
#+LATEX_HEADER: \geometry{margin=25mm}
```

LaTeX classes must be registered in the Emacs configuration before they can be used. The registration maps the class name to a document class declaration and a set of section commands:

```
(with-eval-after-load 'ox-latex
 (add-to-list 'org-latex-classes
  ("scrartcl"
   "\\documentclass[11pt,a4paper]{scrartcl}"
   ("\\section{%s}" . "\\section*{%s}")
   ("\\subsection{%s}" . "\\subsection*{%s}")
   ("\\subsubsection{%s}" . "\\subsubsection*{%s}")))))
```

The PDF compilation command is set through `org-latex-pdf-process`. Using `latexmk` handles multi-pass compilation automatically:

```
(setq org-latex-pdf-process
      '("latexmk -xelatex -quiet %f"))
```

Math in Org is written in LaTeX notation and exported correctly to both LaTeX and HTML:

```
Inline: \langle E = mc^2 \rangle
Display:
\[
\int_a^b f'(x) dx = f(b) - f(a)
\]
```

For HTML math rendering, MathJax or KaTeX is typically used — configured in the `#+HTML_HEAD:` or by setting `org-html-mathjax-options`.

ODT export

The ODT backend (`ox-odt`) produces OpenDocument Text files compatible with LibreOffice and Word. This is Org's path to the `.docx` format: export to ODT, then convert with LibreOffice from the command line:

```
# Export Org to ODT, then convert to DOCX
emacs --batch --eval "(require 'org)" \
  --visit document.org --funcall org-odt-export-to-odt

libreoffice --headless --convert-to docx document.odt
```

ODT export supports custom templates — a `.ott` file (ODT template) that provides the styles the export uses. This is the mechanism for applying a specific visual design to Org's ODT output, analogous to a LaTeX document class.

EPUB export

EPUB export is provided by the `ox-epub` package, which must be installed separately:

```
# In Emacs:  
M-x package-install RET ox-epub RET
```

Once installed and loaded, EPUB export appears in the dispatch menu:

```
#+EPUB_COVER: cover.jpg  
#+EPUB_STYLESHEET: epub.css
```

Export from the command line:

```
emacs --batch \  
  --eval "(require 'org)" \  
  --eval "(require 'ox-epub)" \  
  --visit "document.org" \  
  --funcall org-epub-export-to-epub
```

Pandoc is an alternative path for Org-to-EPUB conversion. Pandoc reads `.org` files natively, so the full Pandoc EPUB pipeline from Chapter 9 is available:

```
pandoc document.org -f org -o output.epub --toc --split-level=1
```

Pandoc's Org reader does not support all Org features (Babel blocks are ignored, some export keywords have no equivalent), but for standard prose documents it is a clean alternative to `ox-epub`.

Babel: literate programming in Org

Babel is Org’s code execution subsystem. It allows source blocks to be executed inline, with results inserted into the document. This is the same capability as Quarto’s code execution, but implemented within Emacs and predating Quarto by over a decade.

A Babel source block:

```
#+BEGIN_SRC python :results output :exports both
import sys
print(f"Python {sys.version}")
print("Typography begins with the alphabet.")
#+END_SRC

#+RESULTS:
: Python 3.12.3
: Typography begins with the alphabet.
```

The `:results output` header argument captures standard output. The `:exports both` argument includes both the source code and the result in the export. Other common header arguments:

- `:results value` — capture the return value rather than stdout
- `:results table` — format a list-of-lists as an Org table
- `:exports code` — export only the code, not the results
- `:exports results` — export only the results, not the code
- `:exports none` — execute but include neither in export
- `:eval never-export` — don’t execute during export (use cached results)
- `:session *name*` — run in a persistent session (preserves state between blocks)
- `:var x=value` — pass a value to the block as a variable
- `:dir /path/` — run the block in a specific directory
- `:tangle filename.py` — extract the code to a file (the “tangling” half of literate programming)

Babel supports dozens of languages: Python, R, shell, Emacs Lisp, C, C++, JavaScript, Haskell, Lua, SQL, LaTeX, Graphviz, Gnuplot, and many more. Languages are enabled in the Emacs configuration:

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '((python . t)
 (shell . t)
 (R . t)
 (latex . t)
 (gnuplot . t)
 (dot . t) ; Graphviz DOT
 (emacs-lisp . t)))
```

Block chaining allows the output of one block to become the input to another:

```
#+NAME: generate-data
#+BEGIN_SRC python :results value
return [(i, i**2) for i in range(1, 11)]
#+END_SRC

#+BEGIN_SRC python :var data=generate-data :results output
for n, sq in data:
    print(f" {n:2d}^2 = {sq:3d}")
#+END_SRC
```

The `:var data=generate-data` header argument passes the results of the `generate-data` block to the second block as the variable `data`. This enables multi-step computational pipelines within a single document.

The tangling system extracts code from Babel blocks to create runnable source files — the other half of Literate Programming in Knuth's original sense:

```
#+BEGIN_SRC python :tangle build.py :exports none
import subprocess

def build_pdf(source):
    """Build a PDF from a Markdown source file."""
```

```

result = subprocess.run(
    ["pandoc", source, "-o", source.replace(".md", ".pdf")],
    capture_output=True
)
return result.returncode == 0
#+END_SRC

```

Running `C-c C-v t` (or `M-x org-babel-tangle`) extracts all tangled blocks to their specified files. The document is simultaneously a piece of human-readable prose explaining the code and the source of the code itself — a genuine literate program.

AUCTeX: LaTeX editing in Emacs

AUCTeX is the Emacs package for LaTeX editing. It provides syntax highlighting, automatic indentation, reference management, smart compilation, and a set of keyboard shortcuts that make writing LaTeX considerably faster. It is the standard LaTeX authoring environment for Emacs users.

Install AUCTeX through Emacs's package system:

```
M-x package-install RET auctex RET
```

A minimal AUCTeX configuration:

```

(use-package auctex
  :defer t
  :hook
  ((LaTeX-mode . LaTeX-math-mode)
   (LaTeX-mode . reftex-mode)
   (LaTeX-mode . flycheck-mode))
  :custom
  (TeX-engine 'xetex)           ; use XeLaTeX
  (TeX-PDF-mode t)             ; produce PDF output
  (TeX-save-query nil)         ; auto-save before compiling
  (TeX-source-correlate-mode t) ; synctex for source-PDF sync

```

```

:config
;; Add latexmk as a compilation command
(add-to-list 'TeX-command-list
'("LatexMk" "latexmk -xelatex %s"
  TeX-run-TeX nil t
  :help "Run latexmk with XeLaTeX"))

```

Key AUCTeX commands:

- C-c C-c — smart compile: determines the right command based on context (LaTeX, BibTeX, view, etc.)
- C-c C-v — view the compiled PDF
- C-c C-e — insert an environment (prompts for environment name, with completion)
- C-c C-s — insert a sectioning command
- C-c C-m — insert a macro
- C-c ~ — toggle math mode
- C-c C-a — compile and view in one command

RefTeX is AUCTeX's companion for cross-reference and citation management. It parses the document structure and bibliography to provide completion for `\ref`, `\cite`, and related commands:

```

(use-package reftex
  :after auctex
  :custom
  (reftex-plug-into-AUCTeX t)
  (reftex-cite-format 'natbib))

```

With RefTeX active: - C-c (— insert a `\label{...}` with completion - C-c) — insert a `\ref{...}` with completion (shows label context) - C-c [— insert a citation with BibTeX key completion

SyncTeX integration — configured through `TeX-source-correlate-mode t` — enables bidirectional navigation between the source file and the PDF. In a compatible PDF viewer (Evince, Skim, Zathura), clicking in the PDF jumps to the corresponding source location; pressing C-c C-v in the source jumps to the corresponding PDF location.

Preview-LaTeX, bundled with AUCTeX, renders mathematical formulas as inline images in the Emacs buffer. This provides immediate visual feedback for complex equations without compiling the full document:

```
C-c C-p C-b  - preview entire buffer
C-c C-p C-s  - preview at point (single formula)
C-c C-p C-c  - clear all previews
```

A practical Org workflow

Org's document production workflow is centred in Emacs, but the output integrates with the same CLI infrastructure as every other tool in this book. A Make-based build for an Org project:

```
DOCUMENT := document
ORG_FILE  := $(DOCUMENT).org
BUILD     := build
EMACS     := emacs --batch

.PHONY: all html pdf odt clean

all: html pdf

$(BUILD):
    mkdir -p $@

$(BUILD)/$(DOCUMENT).html: $(ORG_FILE) | $(BUILD)
    $(EMACS) \
        --eval "(require 'org)" \
        --eval "(setq org-export-with-smart-quotes t)" \
        --visit "$<" \
        --funcall org-html-export-to-html
    mv $(DOCUMENT).html $@

$(BUILD)/$(DOCUMENT).pdf: $(ORG_FILE) | $(BUILD)
    $(EMACS) \
        --eval "(require 'org)" \
        --eval "(setq org-latex-pdf-process \
            '(\"latexmk -xelatex -quiet %f\"))" \
```

```
--visit "$<" \  
--funcall org-latex-export-to-pdf  
mv $(DOCUMENT).pdf $@  
  
clean:  
rm -rf $(BUILD) *.html *.pdf *.tex *.aux *.log
```

The emacs `--batch` approach works without a display and is suitable for CI/CD pipelines. The startup overhead (loading Emacs and its configuration) adds a few seconds to each build, but for document-sized projects this is acceptable.

Org Mode and Emacs represent the deepest end of the CLI typesetting spectrum. The learning investment is real and substantial. For Emacs users who have made that investment, Org's combination of outline editing, spreadsheet tables, Babel code execution, and multi-format export makes it a uniquely capable authoring environment. For everyone else, the tools covered in the preceding chapters cover the same functional ground with a shallower learning curve.

The next chapter examines the tools at the other end of the age spectrum: `groff`, `troff`, and the Unix heritage that predates all of this book's other tools by decades.

groff, troff, and the Unix Heritage

In 1973, Joe Ossanna at Bell Labs typeset the first edition of the Unix Programmer’s Manual using a program he called troff. The document was produced on a Graphic Systems CAT phototypesetter — a machine that exposed characters onto photosensitive paper by spinning a disc of glass type past a light source. The result was the first Unix documentation, set in proportional type, produced by the same computer it documented.

Fifty years later, groff — the GNU reimplement of troff — still ships on every Linux system. It is the tool that formats man pages when you run `man ls` or `man pandoc`. It is smaller and faster than LaTeX, requires no separate installation beyond the system’s standard package set, and for the specific task of writing Unix manual pages, it is the correct and standard tool. For everything else, it is a historical artifact that rewards study but does not reward adoption for new work.

This chapter covers groff from both angles: as the tool you need to understand to write man pages correctly, and as a piece of computing history that illuminates why the tools in the rest of this book are designed the way they are.

The *roff family

The lineage of the *roff tools begins not at Bell Labs but at MIT. In 1964, Jerome Saltzer wrote RUNOFF for the Compatible Time-Sharing System — a program that took a file of text interspersed with formatting commands and produced formatted output. The name was a shorthand for “run it off,” printing slang for producing a copy. Ken Thompson and Bob Morris ported a version of RUNOFF to Unix in 1969 and called it roff.

Joe Ossanna rewrote roff twice. The first rewrite, *nroff* (“new roff”), produced output for character-based terminals and teletype printers, using overstriking and spacing to approximate bold and underline on devices that could not do either. The second rewrite, *troff* (“typesetter roff”), produced output for the CAT phototypesetter with proportional fonts, real point sizes, and genuine typographic quality. Both programs share the same command language — input to *nroff* and *troff* is interchangeable, with *nroff* degrading gracefully on devices that cannot render typeset output.

Ossanna died in 1977. Brian Kernighan rewrote *troff* in C, generalising it to produce output for any device rather than only the CAT. This *device-independent troff* (ditroff) is the direct ancestor of modern *groff*. The GNU Project’s James Clark released *groff* in 1990 as a free software implementation, adding PostScript and PDF output, Unicode support, and various extensions to the macro language.

The *roff command language is built around two concepts*: requests* and *macros*. A request is a two-letter command beginning with a dot at the start of a line that directly instructs the formatter:

```
.ps 11      \" set point size to 11
.vs 14p    \" set vertical spacing to 14 points
.ft I      \" switch to italic font
.br        \" line break
.sp 1      \" one line of vertical space
```

A macro is a named sequence of requests, defined with `.de` and invoked like a request. All the high-level document formatting — headings, paragraphs, footnotes, tables of contents — is built from macros, packaged in *macro packages*: collections of related macros that provide a complete document formatting system.

The major macro packages are:

-man — the man page macros, used for Unix manual pages. Every man page you have ever read was formatted by *groff* with these macros. They are minimal by design: man pages have a fixed structure, and the macros enforce it.

-ms — the manuscript macros, developed at Bell Labs. General-purpose document formatting with support for abstracts, multi-column layout, footnotes, and displays. Historically used for Bell Labs technical memoranda; still useful for simple documents.

-me — Eric Allman’s manuscript macros from Berkeley. Similar in scope to **-ms** but with different syntax and conventions.

-mm — the memorandum macros from AT&T. Common in enterprise Unix environments; follows AT&T’s document formatting conventions.

-mom — Peter Schaffter’s “mom’s own macros,” a modern, full-featured package with a clean English-language interface. The most approachable macro package for new groff documents.

-mdoc — the semantic man page macros from BSD. More structured than **-man**, with macros for specific content types (commands, flags, pathnames, library functions) rather than just presentation. Preferred on FreeBSD and macOS; less common on Linux, which uses **-man**.

The groff command

The `groff` command is the primary interface to the formatter. Its most important flag is `-T`, which selects the output device:

```
groff -T utf8    # formatted text for terminals (nroff mode)
groff -T ps     # PostScript
groff -T pdf    # PDF (groff 1.22+)
groff -T html   # HTML (basic, limited fidelity)
groff -T dvi    # DVI
```

The macro package is selected with `-m`:

```
groff -m man    # -man macros (man pages)
groff -m ms     # -ms macros (manuscripts)
groff -m mom    # -mom macros
```

Shorthand alternatives: `-man`, `-ms`, `-mom` as standalone flags work on many systems.

Preprocessors — programs that process specific markup within a groff file before groff itself sees it — are invoked with single-letter flags:

```
groff -t # run tbl preprocessor (for tables)
groff -e # run eqn preprocessor (for equations)
groff -p # run pic preprocessor (for diagrams)
groff -R # run refer preprocessor (for references)
```

A complete pipeline for a manuscript with tables and equations:

```
groff -ms -t -e document.ms -T pdf > output.pdf
```

The standard workflow for reading man pages:

```
# The normal path: man invokes groff internally
man pandoc

# Read a local man page file
man -l myprogram.1

# Format a man page to PDF
groff -man myprogram.1 -T pdf > myprogram.pdf

# Format to terminal text
groff -man myprogram.1 -T utf8 | less
```

The `-ms` macros for documents

The `-ms` macros provide a practical document formatting system for short to medium documents. They are not competitive with LaTeX for complex work, but for a simple report, a technical memorandum, or a document that must be produced with no dependencies beyond a standard Unix system, they are adequate and fast.

A minimal `-ms` document:

```

.\" article.ms
.\" Format with: groff -ms article.ms -T pdf > article.pdf
.TL
Typography and Typesetting from the CLI
.AU
A. N. Author
.AI
University of Example
.AB
This article demonstrates the ms macro package.
.AE
.
.NH 1
Introduction
.LP
The
.I troff
typesetting system was created at Bell Labs in 1973.
It was used to typeset the first Unix Programmer's Manual.
.
.PP
Paragraphs with first-line indentation use
.CW .PP .
Left-aligned paragraphs without indentation use
.CW .LP .
.
.NH 2
Code examples
.LP
A displayed code example:
.DS
    pandoc input.md -o output.pdf --pdf-engine=xelatex
.DE
.LP
The
.CW .DS / .DE
pair offsets the content and prevents filling (word-wrapping).

```

Key `-ms` macros:

Macro	Purpose
<code>.TL</code>	Title
<code>.AU</code>	Author

Macro	Purpose
.AI	Author institution
.AB / .AE	Abstract begin / end
.NH N	Numbered heading, level N
.SH	Unnumbered section heading
.LP	Left-justified paragraph
.PP	Indented paragraph
.QP	Block quotation paragraph
.DS / .DE	Display (non-flowing) block
.FS / .FE	Footnote start / end
.KS / .KE	Keep together (don't split across pages)
.B	Bold
.I	Italic
.CW	Constant-width (monospace)

Inline font changes use the backslash escape sequences common to all **roff*:

```
Here is \fBbold\fR, \fIitalic\fR, and \f(CWmonospace\fR text.
```

`\fB`, `\fI`, `\fR`, and `\f(CW` switch to bold, italic, roman, and constant-width respectively. The `\fR` restores roman text after any font change. The `\f(CW` form uses the two-character font name `CW` (constant-width), as opposed to the one-character names `B`, `I`, `R`.

Writing man pages

Man pages are the lingua franca of Unix documentation. Every command, system call, library function, file format, and convention used in Unix-like systems is documented in a man page. Writing a correct, complete man page is a non-trivial typographic task — the format has its own conventions, its own layout requirements, and its own reader expectations — but it is also genuinely useful: a well-written man page is often better documentation than a web page, because it is always local, always available, and always consistent.

Structure

A man page has a fixed section order, established by decades of convention:

NAME — the command name and a single-line description, in the form name - description. The hyphen must be an ASCII hyphen, not an em dash. This section is what apropos and man -k search.

SYNOPSIS — the command's calling syntax, using typographic conventions to distinguish mandatory arguments, optional arguments, and repeatable arguments. Square brackets enclose optional elements; ellipses indicate repetition; bold marks command names and flags; italic marks metavariables (placeholders for actual values).

DESCRIPTION — a detailed explanation of what the command does.

OPTIONS — each flag and its effect, typically formatted as a tagged paragraph with the flag in bold and the description indented below.

EXAMPLES — usage examples that demonstrate the command in realistic contexts.

FILES — relevant configuration files, data files, or directories.

ENVIRONMENT — environment variables the command reads.

EXIT STATUS — exit codes and their meanings.

SEE ALSO — related commands and manual pages.

BUGS — known limitations or issues.

AUTHORS — attribution.

Not every section is required. A simple command needs NAME, SYNOPSIS, DESCRIPTION, and OPTIONS at minimum; the others are added as relevant.

The -man macros

The key macros for writing man pages:

`.TH` opens the page and sets its metadata:

```
.TH TYPESET 1 "2024-03-15" "typeset 1.0" "User Commands"
```

Arguments: name, section number, date, source (version/package), and manual title. The section number follows the Unix convention: 1 for user commands, 2 for system calls, 3 for library functions, 5 for file formats, 8 for system administration.

`.SH` begins a section header. Convention dictates all-caps:

```
.SH NAME
.SH SYNOPSIS
.SH DESCRIPTION
```

`.TP` is the tagged paragraph macro — the primary building block of the `OPTIONS` section. The first line following `.TP` becomes the tag (left-justified, in the hanging position); subsequent lines form the description (indented):

```
.TP
.BI \-f "format"
Output format.
One of
.BR html ,
.BR pdf ,
or
.BR epub .
Default is
.BR pdf .
```

Font alternating macros — `.BI`, `.BR`, `.IR`, `.RB`, `.RI`, `.BO` — take their arguments and apply alternating fonts. `.BI` alternates bold and italic, `.BR` alternates bold and roman. This is the standard idiom for the `SYNOPSIS` section, where command names are bold and placeholders are italic:

```
.SH SYNOPSIS
.B typeset
.RB [ \-f
.IR format ]
.RB [ \-o
.IR output ]
.IR file
```

.EX and **.EE** mark example blocks (monospace, non-filling):

```
.PP
Convert a Markdown file to PDF:
.PP
.RS
.EX
typeset article.md
.EE
.RE
```

The **.RS** / **.RE** pair (relative start / relative end) increases the indent for the example block, which is conventional for examples in man pages.

A complete man page

Here is a complete, conventional man page for a hypothetical command:

```
.TH TYPESET 1 "2024-03-15" "typeset 1.0" "User Commands"
.SH NAME
typeset \- produce typeset output from Markdown source
.SH SYNOPSIS
.B typeset
.RB [ \-f
.IR format ]
.RB [ \-o
.IR output ]
.IR file
.SH DESCRIPTION
.B typeset
converts a Markdown source file to a typeset document.
```

It uses Pandoc for conversion and supports HTML, PDF, and EPUB output.

.PP

When

.I file

is

.BR \- ,

standard input is used.

.SH OPTIONS

.TP

.BI \-f " format"

Output format.

One of

.BR html ,

.BR pdf ,

or

.BR epub .

Default is

.BR pdf .

.TP

.BI \-o " file"

Write output to

.I file

instead of the default path.

.TP

.BR \-v " , " \-\-verbose

Print the Pandoc command being executed.

.SH EXAMPLES

Convert a Markdown file to PDF:

.PP

.RS

.EX

typeset article.md

.EE

.RE

.PP

Convert to HTML:

.PP

.RS

.EX

typeset \-f html \-o output.html article.md

.EE

.RE

.SH FILES

.TP

.I ~/.config/typeset/defaults.yaml

```

Default Pandoc options.
.SH SEE ALSO
.BR pandoc (1),
.BR groff (1)
.SH AUTHORS
Written as an example for
.IR "The CLI Typographer" .

```

Several conventions in this example are worth noting. The `\-` escape produces a proper hyphen (minus sign) rather than a hyphen that may be converted to an en dash by `groff`. In SYNOPSIS, each optional flag is wrapped in `\[...]` for the bracket and `flag` for the metavariable. The `.SH SEE ALSO` section references related commands with their section numbers in parentheses, using `.BR` to bold the command name and follow it with `(1)` in roman.

Install and view a man page:

```

# Install to a local man directory
mkdir -p ~/.local/share/man/man1
cp typeset.1 ~/.local/share/man/man1/

# Update the man database
mandb ~/.local/share/man

# Read it
man typeset

```

Pandoc as a man page authoring tool

Writing man pages directly in `groff` syntax is the correct approach for complex pages with many formatting details. For simpler pages, Pandoc's man output format offers an alternative: write the man page in Markdown with appropriate metadata, and let Pandoc generate the `troff` source.

```

---
title: TYPESET
section: 1
date: 2024-03-15
header: User Commands
footer: typeset 1.0
---

# NAME

typeset - produce typeset output from Markdown source

# SYNOPSIS

**typeset** [-f format] [-o output] file*

# DESCRIPTION

**typeset** converts a Markdown file to a typeset document.

# OPTIONS

-f format
:   Output format: html, pdf, or epub. Default: pdf.

-o file*
:   Write output to file*.

# SEE ALSO

**pandoc(1), groff(1)

```

```
pandoc mycommand.md -t man > mycommand.1
```

Pandoc translates Markdown's structure to the corresponding man macros: # headings become .SH sections, definition lists (: definition) become .TP tagged paragraphs, bold and italic become the appropriate font switches.

The Pandoc-generated source is readable and correct, though it uses \f[B] and \f[I] for font changes rather than the more traditional .B and .I macros — both are valid groff, and both render identically.

The reverse path also works: Pandoc reads man format and converts it to HTML, Markdown, or any other output:

```
# Read a man page and convert to HTML documentation
pandoc /usr/share/man/man1/ls.1.gz -f man -t html -s -o ls.html

# Convert your own man page to Markdown (for a README)
pandoc mycommand.1 -f man -t markdown -o USAGE.md
```

This round-trip capability is useful when you want to maintain documentation in man format (for proper Unix packaging) and also publish it as HTML or as a README.

groff for documents today

Groff's practical role in contemporary document production is narrow but real. It is the tool for three specific tasks:

Writing man pages. This is groff's continuing primary purpose. Every Linux and Unix system expects man pages in troff format. Every package manager and documentation system knows how to install and render them. There is no better alternative for man pages that will be distributed with system software.

Reading and converting existing troff documents. Any document in groff's input format can be converted to PDF, HTML, or text. For organisations with legacy technical documentation in ms or mm format, groff is the path to modernisation without full re-authoring.

Simple documents with zero dependencies. A groff installation is part of every Linux system's base package set. A LaTeX installation is not. For shell scripts that need to produce formatted output, or for documents that must be buildable on any Unix system without additional packages, groff provides a complete formatter that is always available.

For general document production — articles, reports, books, documentation — the tools covered in earlier chapters are better choices. LaTeX

provides higher typographic quality and a vastly larger package ecosystem. Pandoc provides format flexibility. Typst provides modern syntax and fast compilation. Groff has none of these advantages for general work. Its virtue is its universality: it is always there, it is always the same, and for man pages, it is exactly right.

The next chapter completes Part III with the tools that generate diagrams and figures from the command line — TikZ, Graphviz, gnuplot, Mermaid, and the others that produce the visual content that documents contain alongside their text.

Diagrams and Figures from the CLI

Text alone cannot do everything a technical document needs. Flowcharts, architecture diagrams, data plots, circuit schematics, mathematical figures, state machines, dependency graphs — these are not decorations. They carry information that prose conveys poorly, and they do so through visual relationships that the reader can grasp at a glance. For the CLI typographer, the question is not whether to include figures, but how to produce them without reaching for a GUI application.

The answer is a set of text-based diagram tools: programs that take a description of what the diagram should contain and produce an image. The description might be a graph in DOT language, a script in Python, a drawing in TikZ, or a flowchart in Mermaid syntax. In every case, the diagram is a file that can be committed to version control, diffed, reviewed, and automated — not a binary blob dropped from a drawing application.

This chapter covers the main tools and their niches, the formats that work best for each output medium, and the workflows for integrating generated figures into Markdown-first, Typst, and LaTeX compatibility workflows.

Choosing the right format

Before choosing a diagram tool, choose the output format. The diagram's purpose in the document determines which format is correct, and the format constrains which tools are appropriate.

SVG is the right format for diagrams in HTML output. It is resolution-independent (the diagram looks sharp at any screen density), its text content is selectable and searchable, and it integrates naturally with CSS.

The file size is typically small for line diagrams. SVG can also be used in PDF output when converted to PDF format first, or embedded directly in PDF by some engines.

PDF is the right format for diagrams in print-oriented PDF workflows. TikZ produces PDF natively (as part of the document); Graphviz and other tools can output PDF directly; SVG files can be converted to PDF with `pdftocairo`, `inkscape`, or `rsvg-convert`. Embedded PDFs in Typst or LaTeX are rendered at full resolution.

PNG at adequate resolution (150–300 dpi for screen, 300–600 dpi for print) works for both HTML and PDF output. It is raster rather than vector, so it does not scale cleanly, but it is universally supported and is the right choice for diagrams with complex gradients or photographic content that SVG cannot represent well.

EPS (Encapsulated PostScript) is the older format for LaTeX figures. It is still accepted by some publishers and older LaTeX workflows. Graphviz can output EPS; `pdftocairo -eps` converts PDFs to EPS. For new work, PDF is preferable.

The practical default for CLI diagram production: generate SVG for HTML output, PDF for print-oriented PDF backends such as Typst or LaTeX. If a tool produces only one format, convert as needed.

Graphviz: graphs and networks

Graphviz is the standard tool for automatically laying out graphs — directed and undirected networks of nodes connected by edges. It is appropriate whenever the structure matters more than the precise position: dependency graphs, state machines, call graphs, organisational charts, network topology, data flow diagrams.

Install Graphviz on Debian/Ubuntu: `apt install graphviz`. It provides six layout programs, each suited to different graph structures.

The DOT language

Graphviz diagrams are written in DOT, a declarative language that describes nodes, edges, and their attributes:

```
digraph pipeline {
  rankdir=LR;
  node [shape=box, style=rounded, fontname="sans-serif", fontsize=11];

  md    [label="Markdown\nsource"];
  pandoc [label="Pandoc", shape=ellipse];
  pdf   [label="PDF"];
  html  [label="HTML"];
  epub  [label="EPUB"];

  md -> pandoc;
  pandoc -> pdf;
  pandoc -> html;
  pandoc -> epub;
}
```

`digraph` introduces a directed graph (arrows); `graph` introduces an undirected graph (lines). `rankdir=LR` sets the layout direction to left-to-right; the default is top-to-bottom. `node [. . .]` sets default attributes for all nodes; individual nodes can override these.

Compile to SVG, PDF, or PNG:

```
dot -Tsvg pipeline.dot -o pipeline.svg
dot -Tpdf pipeline.dot -o pipeline.pdf
dot -Tpng -Gdpi=150 pipeline.dot -o pipeline.png
```

Layout engines

The `dot` program uses a hierarchical layout appropriate for directed graphs with a natural direction (top-to-bottom or left-to-right flows). For other graph structures, different layout engines are more appropriate:

```

neato # spring model – good for undirected graphs and networks
fdp   # force-directed – similar to neato, larger graphs
sfdp  # scalable force-directed – very large graphs
twopi # radial layout – circular arrangement around a root node
circo # circular layout – regular ring arrangement

```

All accept the same DOT input and produce the same output formats. The choice of engine is a matter of what looks best for the specific graph structure.

Styling

Graphviz supports a rich set of visual attributes. The key attributes for professional-quality diagrams:

Node shapes: box, ellipse, circle, diamond, doublecircle, plaintext, note, tab, folder, record. The record shape creates structured multi-field nodes:

```

node [shape=record, fontname="sans-serif", fontsize=10];
font [label="{ Font file | { Family | Style } | { OTF | TTF } }"];

```

Colors and fills: Use named colors (lightblue, lightyellow) or hex values. Set style=filled to enable fill colour, fillcolor="#e8f4fb" for the fill, and color="#666666" for the border:

```

node [style=filled, fillcolor="#f0f8ff", color="#999999"];

```

Subgraphs and clusters: Grouping related nodes in a subgraph cluster_name { } draws a box around them:

```

subgraph cluster_outputs {
  label="Outputs";
  style=dashed;
  color=gray;
  pdf; html; epub;
}

```

Transparency and backgrounds: Set `bgcolor="transparent"` on the graph for a transparent background, which integrates cleanly when embedded in documents with non-white backgrounds.

When not to use Graphviz

Graphviz is for graphs: nodes and edges. It is the wrong choice for diagrams where spatial position or layout geometry is important — flowcharts where specific branching positions matter, architectural diagrams where the spatial relationships convey meaning, or any diagram where the automatic layout produces results that do not match the intended structure. For those cases, TikZ (for LaTeX) or a manual SVG gives more control.

TikZ: programmatic drawing in LaTeX

TikZ (a recursive acronym for “TikZ ist kein Zeichenprogramm” — TikZ is not a drawing program) is the LaTeX package for programmatic vector graphics. It is the natural choice for diagrams in LaTeX documents: the diagram is typeset as part of the document, uses the same fonts, and can reference LaTeX commands, counters, and defined lengths.

TikZ is comprehensive and complex. The TikZ manual runs to over 1200 pages. This section covers the patterns needed for technical documentation; the rest can be learned by example and reference.

Basic structure

TikZ diagrams live inside a `tikzpicture` environment:

```
\usepackage{tikz}
\usetikzlibrary{arrows.meta, positioning, shapes.geometric}

\begin{tikzpicture}[
```

```

node distance=2cm,
box/.style={
  rectangle, rounded corners=4pt, draw=gray!60, line width=0.5pt,
  minimum width=2.5cm, minimum height=0.8cm, text centered,
  font=\small\sffamily
}
]
\node[box, fill=blue!10] (md) {Markdown};
\node[box, fill=orange!15, right=of md] (pandoc) {Pandoc};
\node[box, fill=green!10, right=of pandoc] (pdf) {PDF};

\draw[-{Stealth}] (md) -- (pandoc);
\draw[-{Stealth}] (pandoc) -- (pdf);
\end{tikzpicture}

```

The [...] after `\begin{tikzpicture}` sets defaults for the whole diagram. The `node distance=2cm` sets the default spacing for the positioning library's `above=of`, `right=of`, and `below=of` placement commands. Named styles (`.style={...}`) define reusable attribute sets.

Standalone diagrams

For diagrams that need to be used outside LaTeX — in Pandoc HTML output, in Typst, or as standalone figures — compile them with the `standalone` class:

```

\documentclass{standalone}
\usepackage{tikz}
\usetikzlibrary{arrows.meta, positioning}

\begin{document}
\begin{tikzpicture}[...]
  % diagram content
\end{tikzpicture}
\end{document}

```

The `standalone` class sizes the PDF page tightly to the diagram content. Compile and convert to SVG:

```
pdflatex diagram.tex
pdftocairo -svg diagram.pdf diagram.svg
```

pdftocairo from the poppler-utils package is the cleanest SVG output from TikZ. The result is a precise SVG with correct fonts and geometry.

TikZ libraries

TikZ's functionality is extended by libraries, loaded with `\usetikzlibrary{}`:

- `arrows.meta` — modern arrowhead styles (`{Stealth}`, `{LaTeX}`, `{Circle}`)
- `positioning` — relative node placement (`right=2cm of nodename`)
- `shapes.geometric` — additional node shapes (diamond, trapezium, cylinder)
- `fit` — nodes that automatically size to contain other nodes
- `calc` — coordinate arithmetic (`$(a)!0.5!(b)$` for midpoints)
- `matrix` — table-like layouts of nodes
- `decorations` — decorative path effects (wavy lines, braces)
- `shadows` — drop shadow effects (use sparingly)

For PGFPlots (data plotting within LaTeX), the separate pgfplots package provides `\begin{axis}` environments for function plots, scatter plots, bar charts, and more, all typeset with LaTeX fonts.

Mermaid: diagrams from text in web contexts

Mermaid is a JavaScript library and CLI tool that renders diagrams from a text description. It is particularly well-suited for Markdown documentation and web contexts, where diagrams can be embedded as Mermaid source blocks and rendered client-side by a JavaScript include, or pre-rendered to SVG by the CLI tool.

Mermaid supports a wide range of diagram types with a simple, readable syntax:

Flowcharts:

```
flowchart LR
  A[Markdown] --> B{Pandoc}
  B --> C[PDF]
  B --> D[HTML]
  B --> E[Epub]
```

Sequence diagrams:

```
sequenceDiagram
  Author->>Make: make pdf
  Make->>Pandoc: pandoc input.md -o output.pdf
  Pandoc->>LaTeX: generate .tex
  LaTeX-->>Pandoc: output.pdf
  Pandoc-->>Make: build complete
  Make-->>Author: output.pdf ready
```

Class diagrams, entity-relationship diagrams, state diagrams, Gantt charts, and several others are also supported.

CLI generation

The Mermaid CLI (`mmdc`) converts Mermaid source to SVG, PNG, or PDF:

```
npm install -g @mermaid-js/mermaid-cli

# Generate SVG
mmdc -i diagram.mmd -o diagram.svg

# Generate PNG at specific size
mmdc -i diagram.mmd -o diagram.png -w 800 -H 600

# With a custom CSS theme
mmdc -i diagram.mmd -o diagram.svg --cssFile theme.css
```

Mermaid CLI requires a Chrome or Chromium browser for rendering. On headless servers, Puppeteer's bundled Chromium is used. The initial setup involves installing Puppeteer's Chrome dependency:

```
npx puppeteer browsers install chrome-headless-shell
```

Mermaid in GitHub and documentation platforms

Most documentation platforms — GitHub, GitLab, Notion, Confluence, and many static site generators — render Mermaid blocks natively. In these contexts, no pre-processing is needed: write the Mermaid syntax in a fenced code block and the platform renders it.

For Pandoc-based HTML output, Mermaid diagrams can be rendered by including the Mermaid JavaScript library in the HTML template and using fenced code blocks with the `mermaid` class — Mermaid detects and renders them automatically:

```
<!-- In Pandoc HTML template or --include-after-body -->  
<script src="https://cdn.jsdelivr.net/npm/mermaid/dist/mermaid.min.js"></script>  
↪ script>  
<script>mermaid.initialize({startOnLoad: true});</script>
```

For PDF output, the CLI pre-render to SVG is the practical path.

D2: a modern alternative to DOT

D2 is a newer diagram scripting language with a cleaner syntax than DOT and built-in support for multiple layout engines, including TALA (a proprietary layout engine optimised for software architecture diagrams). It is worth knowing for new projects that do not need to integrate with existing Graphviz tooling.

```
# Pipeline diagram in D2
direction: right

md: Markdown {shape: page}
pandoc: Pandoc {shape: oval}
outputs: {
  pdf: PDF {shape: document}
  html: HTML {shape: document}
  epub: EPUB {shape: document}
}

md -> pandoc
pandoc -> outputs.pdf: via LaTeX
pandoc -> outputs.html
pandoc -> outputs.epub
```

```
d2 diagram.d2 output.svg
d2 --layout=elk diagram.d2 output.svg # use ELK layout engine
d2 --theme=200 diagram.d2 output.svg # numbered themes
```

D2 is still maturing, but its syntax is noticeably more readable than DOT for complex diagrams, and its layout results for software architecture diagrams tend to be better than Graphviz’s default dot layout.

Data plots with gnuplot and Python

Technical documents often contain charts generated from data — performance benchmarks, experimental results, statistical summaries. These need to be generated programmatically from data sources rather than drawn manually.

gnuplot is the traditional Unix tool for this. It produces high-quality plots for printed documents, with excellent PostScript and PDF output:

```
# reading-speeds.gnuplot
set terminal svg size 600,400 font "sans-serif,11"
set output "reading-speeds.svg"
```

```
set xlabel "Point size"
set ylabel "Words per minute"
set key top left
set border 3
set tics nomirror
set style data linespoints

plot "data.txt" using 1:2 title "Garamond", \
     "data.txt" using 1:3 title "Palatino", \
     "data.txt" using 1:4 title "Times"
```

```
gnuplot reading-speeds.gnuplot
```

gnuplot's syntax is idiosyncratic but powerful. For publication-quality academic figures, it remains the tool of choice in many scientific communities.

Python with matplotlib offers a more programmable alternative with a larger ecosystem of extensions. For complex data manipulation before plotting, for interactive exploration before committing to a final figure, or in projects already using Python for computation, matplotlib is often the practical choice:

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np

# Use a style that matches your document's aesthetics
mpl.rcParams.update({
    'font.family':      'serif',
    'font.serif':      ['EB Garamond', 'Georgia'],
    'font.size':        11,
    'axes.spines.top':  False,
    'axes.spines.right': False,
    'figure.dpi':       150,
})

fig, ax = plt.subplots(figsize=(6, 4))

typefaces = ['Garamond', 'Palatino', 'Times']
```

```

sizes = [9, 11, 14]
speeds = [[220, 255, 268], [225, 258, 271], [218, 250, 265]]

x = np.arange(len(typefaces))
for i, (size, vals) in enumerate(zip(sizes, zip(*speeds))):
    ax.bar(x + i*0.25, vals, 0.25, label=f"{size}pt")

ax.set_xticks(x + 0.25)
ax.set_xticklabels(typefaces)
ax.set_ylabel("Words per minute")
ax.legend(frameon=False, title="Size")

# Save as SVG for HTML, PDF for print workflows
fig.savefig("reading-speeds.svg", format="svg", bbox_inches="tight")
fig.savefig("reading-speeds.pdf", format="pdf", bbox_inches="tight")

```

The rcParams settings at the top establish document-consistent typography for the plot. Matching the plot's font to the document's body font is the single most important step toward figures that look like they belong in the document rather than having been imported from a different context.

Integrating diagrams in documents

In Pandoc Markdown

For HTML output, SVG and PNG figures work directly:

```
![[Pipeline diagram](figures/pipeline.svg){width=80%}]
```

The `{width=80%}` attribute is translated to `style="width:80%"` in HTML and `[width=0.8\textwidth]` in LaTeX (via Pandoc's automatic translation).

For PDF output via a print-oriented backend, vector figures should be in PDF format. The simplest approach is to maintain both versions and use a Lua filter to select the right format:

```
-- figures-filter.lua: use .pdf for LaTeX, .svg for HTML
function Image(e1)
  if FORMAT:match("latex") or FORMAT:match("pdf") then
    e1.src = e1.src:gsub("%.svg$", ".pdf")
  end
  return e1
end
```

```
pandoc input.md --lua-filter=figures-filter.lua -o output.pdf
```

In LaTeX directly

In native LaTeX, figures use the figure environment and `\includegraphics` from the `graphicx` package:

```
\usepackage{graphicx}

\begin{figure}[htbp]
  \centering
  \includegraphics[width=0.8\textwidth]{figures/pipeline}
  \caption{The Pandoc conversion pipeline}
  \label{fig:pipeline}
\end{figure}
```

Omitting the file extension from `\includegraphics` lets LaTeX find the best available format: it prefers PDF, then EPS, then PNG. This means you can have both `pipeline.pdf` and `pipeline.png` and LaTeX will use the PDF automatically.

In Typst

Typst includes figures with the `#figure` function:

```
#figure(
  image("figures/pipeline.svg", width: 80%),
  caption: [The Pandoc conversion pipeline],
) <fig-pipeline>
```

Typst supports SVG and common raster formats natively, without needing format-specific conversion.

A build script for figures

When a document contains many generated figures, a Makefile target or dedicated build script manages generation:

```
# figures.mk - include in main Makefile
FIGURES_DIR := figures
DOT_SOURCES := $(wildcard $(FIGURES_DIR)/*.dot)
DOT_SVGs    := $(DOT_SOURCES:.dot=.svg)
DOT_PDFS    := $(DOT_SOURCES:.dot=.pdf)

TIKZ_SOURCES := $(wildcard $(FIGURES_DIR)/*.tex)
TIKZ_PDFS    := $(TIKZ_SOURCES:.tex=.pdf)
TIKZ_SVGs    := $(TIKZ_SOURCES:.tex=.svg)

.PHONY: figures figures-clean

figures: $(DOT_SVGs) $(DOT_PDFS) $(TIKZ_PDFS) $(TIKZ_SVGs)

$(FIGURES_DIR)/%.svg: $(FIGURES_DIR)/%.dot
    dot -Tsvg $< -o $@

$(FIGURES_DIR)/%.pdf: $(FIGURES_DIR)/%.dot
    dot -Tpdf $< -o $@

$(FIGURES_DIR)/%.pdf: $(FIGURES_DIR)/%.tex
    cd $(FIGURES_DIR) && \
        pdflatex -interaction=nonstopmode $(notdir $<) >/dev/null 2>&1

$(FIGURES_DIR)/%.svg: $(FIGURES_DIR)/%.pdf
    pdftocairo -svg $< $@

figures-clean:
    rm -f $(DOT_SVGs) $(DOT_PDFS) $(TIKZ_PDFS) $(TIKZ_SVGs)
```

Including this in the main Makefile with `include figures.mk` and adding `figures` as a prerequisite of the main build targets ensures figures are always up to date.

Part III ends here. We have surveyed all the major CLI typesetting tools — LaTeX, Typst, Quarto, Emacs and Org Mode, groff, and the diagram tools that complement them all. Part IV now turns from tools to documents: a gallery of real examples, each built from scratch using the tools and techniques developed in Parts I through III.

Document Gallery: Real Examples

Letters and Correspondence

Part IV turns from tools to documents. The next five chapters produce real documents — letters, résumés, articles, presentations, and books — using the tools and techniques from Parts I through III. Each chapter covers one document type, examines what it requires typographically, and provides working examples in at least two tool pipelines.

The letter is the oldest document type in continuous use. A formal business letter from 2024 shares its basic structure with a formal letter from 1924: sender’s address, date, recipient’s address, salutation, body, complimentary close, signature. The typographic conventions are stable and well-understood, which makes the letter an ideal first document in this gallery — the requirements are clear, the expected output is recognizable, and the implementation illuminates techniques that scale to more complex documents.

This chapter covers three variants: the formal business letter (a submission letter to a journal), the cover letter with a personal letterhead (for job applications), and mail merge (generating one letter per recipient from a CSV file). The general pattern follows the rest of the book: keep the source in Markdown when multiple formats matter, prefer Typst for PDF-only output, and treat specialised postal templates as edge cases rather than the default path.

What a letter requires

A letter’s typographic requirements are modest but precise. The key elements:

Address alignment varies by convention. British and international practice uses *full block* format: every element left-aligned, with blank lines

separating each block (sender address, date, recipient address, salutation, body, close). American practice is similar but uses a colon after the salutation rather than a comma. Older *semi-block* format uses a right-aligned sender block and date, with indented first lines in body paragraphs.

Spacing in a letter is vertical — the blank lines between elements carry as much meaning as the text. The date must be visually separated from the recipient address; the salutation must be separated from the body. This vertical rhythm is what makes a letter look like a letter rather than a document that happens to contain an address.

Font selection for correspondence should feel neutral and professional. Serif typefaces — Garamond, Palatino, Charter, or Source Serif — suit formal business correspondence. A humanist sans-serif is acceptable for contemporary contexts, particularly for letters in creative fields.

Page rules: letters are single-spaced with a blank line between paragraphs. Body text is not indented (in full block format). Headers and footers are typically absent; page numbers appear only when the letter runs to multiple pages, and then only from the second page.

Typst as the default formal-letter template

For PDF-only correspondence, Typst is the cleaner default. The structure of a formal letter maps directly to a Typst function: sender block, date, recipient block, subject, salutation, body, close, and optional enclosure line.

```
#let formal-letter(  
  from-name: "",  
  from-address: "",  
  from-email: none,  
  to-name: "",  
  to-address: "",  
  date: none,  
  subject: none,  
  opening: "Dear Sir or Madam,",  
  closing: "Yours faithfully,",  
  enclosures: none,
```

```

body,
) = {
  set page(paper: "a4", margin: (top: 25mm, bottom: 30mm, left: 30mm,
    ↪ right: 25mm))
  set text(font: "EB Garamond", size: 11pt)

  align(right)[
    #from-name\
    #from-address
    #if from-email != none [#raw(from-email)]
  ]
  v(1.5em)
  align(right)[#date]
  v(2em)

  #to-name\
  #to-address
  v(1.5em)

  if subject != none [*Subject:* #subject #v(0.8em)]

  #opening
  v(0.8em)
  #body
  v(1.5em)
  #closing
  v(3em)
  #from-name

  if enclosures != none [
    v(1.2em)
    *Enclosures:* #enclosures
  ]
}

```

That covers the typographic logic most letters actually need. The sender block is right-aligned, the address blocks are vertically separated, and the body remains plain prose.

Postal-compliance edge cases

The one place older letter systems still matter is postal compliance: window-envelope positioning, fold marks, and organisation-specific stationery rules. If a department or institution already has a locked-down letter system for those cases, treat it as a compatibility path. For ordinary professional correspondence, that should be the exception rather than the chapter's organising model.

Cover letters with a personal header

A cover letter for a job application has different conventions from a formal business letter. It is more personal, often contains design elements that reflect the applicant's aesthetic judgment, and typically uses a custom letterhead rather than a rigid institutional layout.

In Typst, that is simply a different header function:

```
#let letterhead(name, details) = [
  #align(center)[
    #text(font: "Source Sans 3", weight: "bold", size: 14pt)[#upper(name)]
    #v(4pt)
    #text(size: 9pt, fill: luma(110))[#details]
    #v(6pt)
    #line(length: 80%, stroke: 0.4pt + luma(180))
  ]
  #v(1.4em)
]
```

Used in a cover letter:

```
#letterhead(
  "A. N. Author",
  [742 Evergreen Terrace | Springfield, SS 12345 |
  ↪ #link("mailto:author@example.edu")[author@example.edu]],
)
#align(right)[15 March 2024]
```

```
#v(1em)
```

```
Dr Sarah Mitchell\  
Head of Typography\  
Design Institute\  
London, WC2A 2AE
```

```
#v(1.5em)
```

```
Dear Dr Mitchell,
```

The body remains ordinary prose. The header is the only distinctive design element, which is exactly where the applicant's visual judgment should appear.

A Markdown source for flexible formats

For correspondence that needs to be produced in multiple formats — PDF for printing, HTML for email, ODT for editing — writing the letter in Pandoc Markdown with a custom template separates the content from the format entirely. This is the canonical approach when the same letter may need to travel through several systems.

The letter body in Markdown, with metadata:

```
---  
from-name: "A. N. Author"  
from-address: "742 Evergreen Terrace, Springfield, SS 12345"  
from-email: "author@example.edu"  
to-name: "The Editor"  
to-company: "Journal of Typographic Practice"  
to-address: "1 Publisher's Row, London, EC1A 1AA"  
date: "15 March 2024"  
subject: "Manuscript Submission"  
opening: "Dear Editor,"  
closing: "Yours sincerely,"  
---  
  
I am writing to submit the manuscript "On the Typographic Quality  
of CLI-Produced Documents" for consideration in the Journal of  
Typographic Practice.
```

The manuscript presents a systematic comparison of documents produced using command-line tools against those produced using graphical desktop publishing applications.

I would be grateful for your consideration.

For PDF output, a Typst template can read the same metadata and keep the formatting logic out of the source:

```
#let render(meta, body) = formal-letter(
  from-name: meta.at("from-name"),
  from-address: meta.at("from-address"),
  from-email: meta.at("from-email", default: none),
  to-name: meta.at("to-name"),
  to-address: meta.at("to-company") + "\n" + meta.at("to-address"),
  date: meta.at("date"),
  subject: meta.at("subject", default: none),
  opening: meta.at("opening"),
  closing: meta.at("closing"),
  body,
)
```

Render to PDF through the Typst path:

```
pandoc letter.md --template=letter.typ --pdf-engine=typst -o letter.pdf
```

Render to HTML with an equivalent HTML template for the same letter in email-friendly format, or to plain text for quick review. The content never needs to change; only the template changes with the output format.

Mail merge: one letter per recipient

Mail merge — generating a personalised letter for each entry in a list — is a natural extension of the template approach. The same Pandoc

command that renders a letter from YAML metadata can be driven by a CSV file, producing one PDF per row.

The recipient list `recipients.csv`:

```
name,title,company,address,city,role,ref
"Dr Sarah Mitchell","Dr","Design Institute","1 Design Square","London WC2A
↳ 2AE","Senior Document Architect","DA-2024-03"
"Prof James Chen","Prof","Institute of Print","24 Press Lane","Edinburgh
↳ EH1 2AB","Technical Writer","TW-2024-07"
"Ms Rebecca Okonkwo","Ms","Typesetting Ltd","56 Compositor
↳ Street","Bristol BS1 3CD","Document Engineer","DE-2024-12"
```

A Python script that reads the CSV and calls Pandoc for each row:

```
#!/usr/bin/env python3
"""mailmerge.py: generate one PDF per recipient from CSV + Pandoc
↳ template."""

import csv
import subprocess
import sys
from pathlib import Path

def sanitize(name):
    return name.replace(' ', '_').replace('.', '').replace(',', '')

def generate(template, typst_tmpl, recipient, output_path):
    cmd = ['pandoc', template,
           '--template', typst_tmpl,
           '--pdf-engine', 'typst',
           '-o', str(output_path)]
    for key, value in recipient.items():
        cmd.extend(['-M', f'{key}={value}'])
    return subprocess.run(cmd, capture_output=True).returncode == 0

def main():
    csv_file = sys.argv[1] if len(sys.argv) > 1 else 'recipients.csv'
    template = sys.argv[2] if len(sys.argv) > 2 else 'letter.md'
    output_dir = Path(sys.argv[3] if len(sys.argv) > 3 else 'output')
    typst_tmpl = 'letter.typ'

    output_dir.mkdir(parents=True, exist_ok=True)
```

```

success = failures = 0

with open(csv_file, newline='', encoding='utf-8') as f:
    for row in csv.DictReader(f):
        name = row.get('name', 'recipient')
        out = output_dir / f"letter-{sanitize(name)}.pdf"
        print(f" {'OK' } if generate(template, typst_tmpl, row, out)
        ↪ else 'FAIL': {name}")
        success += 1

print(f"\n{success} letters generated in {output_dir}/")

if __name__ == '__main__':
    main()

```

Run the merge:

```
python3 mailmerge.py recipients.csv letter.md output/
```

```

OK  : Dr Sarah Mitchell
OK  : Prof James Chen
OK  : Ms Rebecca Okonkwo

```

3 letters generated in output/

Each recipient's data is passed to Pandoc via `-M key=value` flags, which override and supplement the metadata in the template Markdown file. The name, title, company, address, city, role, and ref fields from the CSV row are injected into the template as metadata variables, where the Typst template reads them and lays out the final PDF.

For a pure-shell alternative without Python:

```

#!/bin/sh
tail -n +2 recipients.csv | while IFS=, read name title company address
↪ city role ref; do
    name=$(printf '%s' "$name" | tr -d ' ')
    title=$(printf '%s' "$title" | tr -d ' ')
    slug=$(printf '%s' "$name" | tr ' ' '_' | tr -d '.')

```

```

pandoc letter.md \
  --template=letter.typ \
  --pdf-engine=typst \
  -M "name=$name" -M "title=$title" \
  -M "company=$(printf '%s' "$company" | tr -d "'')" \
  -M "address=$(printf '%s' "$address" | tr -d "'')" \
  -M "city=$(printf '%s' "$city" | tr -d "'')" \
  -M "role=$(printf '%s' "$role" | tr -d "'')" \
  -M "ref=$(printf '%s' "$ref" | tr -d "'')" \
  -o "output/letter-$(slug).pdf" 2>/dev/null \
  && echo "OK: $name" || echo "FAIL: $name"
done

```

The shell approach is simpler but less robust for CSV fields that contain commas or newlines within quoted values. Python's `csv` module handles these edge cases correctly; the shell approach requires that no field value contains a comma or newline.

The Typst letter template

For projects using Typst as the primary tool, the same letter can be written as a Typst document. For PDF-only correspondence this is usually the cleaner default. The advantage is that the template is written in Typst's native function syntax, which is more readable and easier to modify:

```

#let letter(
  from-name: "",
  from-address: "",
  from-email: none,
  to-name: "",
  to-address: "",
  date: none,
  subject: none,
  opening: "Dear Sir or Madam,",
  closing: "Yours faithfully,",
  body
) = {

```

```

set page(paper: "a4",
  margin: (top: 25mm, bottom: 30mm, left: 30mm, right: 25mm))
set text(font: "EB Garamond", size: 11pt)
set par(justify: true)

// Sender block
align(right)[
  #from-address \
  #if from-email != none { raw(from-email) }
]
v(0.5em)
align(right, if date != none { date } else {
  datetime.today().display("[day] [month repr:long] [year]")
})
v(2em)

// Recipient
to-name \
to-address
v(2em)

if subject != none { strong("Subject: " + subject); v(0.5em) }

opening
v(0.5em)
body
v(1.5em)
closing
v(3em)
from-name
}

// Using the template
#show: letter.with(
  from-name: "A. N. Author",
  from-address: "742 Evergreen Terrace\nSpringfield, SS 12345",
  from-email: "author@example.edu",
  to-name: "The Editor",
  to-address: "Journal of Typographic Practice\n1 Publisher's Row\nLondon,
↵ EC1A 1AA",
  date: "15 March 2024",
  subject: "Manuscript Submission",
  opening: "Dear Editor,",
  closing: "Yours sincerely,",
)

```

```
I am writing to submit the manuscript "On the Typographic Quality
of CLI-Produced Documents" for consideration in the Journal of
Typographic Practice.
```

```
The manuscript presents a systematic comparison of documents
produced using command-line tools. I would be grateful for your
consideration.
```

```
typst compile letter.typ
```

Typst does not yet offer the same built-in postal-specialty ecosystem as older institutional letter workflows. For letters that must fit a windowed envelope precisely or satisfy a rigid stationery standard, an inherited template may still be useful.

Choosing the approach

The right tool for a letter depends on how the letter will be used:

For a one-off formal letter where typographic quality matters and the document will be printed: Typst is the first choice for a new PDF-only workflow. Keep legacy postal templates in reserve only for cases where window-envelope positioning or institutional stationery requirements are non-negotiable.

For a template-driven letter produced across multiple formats: the Pandoc or Quarto approach separates content from presentation cleanly, enables multiple output formats from one source, and keeps the letter content in Markdown where it is easy to read and edit.

For mail merge producing many letters from a data file: the Python + Pandoc approach handles the CSV correctly, produces one file per recipient, and requires no specialised mail merge infrastructure.

For Typst-based projects: the native Typst function approach is clean and readable, with fast compilation and easy modification, and should

be considered the default PDF path unless an inherited postal template blocks it.

The letter is a simple document, but it illustrates the template-and-separation-of-concerns pattern that scales to the more complex documents in the next chapters. The résumé, covered in Chapter 19, extends these patterns significantly.

Résumés and CVs

The résumé and the curriculum vitae are, by definition, documents about the person who produces them. This creates a dynamic unlike most other document types: the author is their own client, the typographic decisions reflect on the author's taste and judgment, and the document will be read by people who are simultaneously evaluating its content and its form. A résumé that is well-typeset signals something about the person it describes. A résumé that is poorly typeset — inconsistent spacing, mismatched fonts, rivers of white space — signals something too.

This chapter covers three variants: the single-column professional résumé, the two-column layout used in design and technology contexts, and the academic CV with a publications section. The default workflow should be to maintain the content in Markdown and render it through Pandoc or Quarto to the formats required by the application process. For PDF output, Typst is often the cleanest backend because its templates remain readable and maintainable. LaTeX is still useful for highly customised layouts and publication-heavy academic CVs, but it should be treated as a backend-specific path rather than the default starting point.

What a CV requires typographically

Before choosing a backend, it is worth establishing what a CV actually needs to do.

A CV must be **scannable**. Hiring managers and admissions committees spend thirty seconds to two minutes on an initial read. The document structure must communicate instantly: where are the jobs, where is the

education, how long was each role. This is a hierarchy problem — the same problem of making structure visible that was addressed in Chapter 2 — but with the added constraint that the document must communicate quickly under time pressure.

A CV must **fit the expected format**. A one-page résumé is conventional for most industry applications; two pages for more senior roles. An academic CV is typically multiple pages and grows over a career as publications accumulate. Deviating from the expected format without a reason creates friction.

A CV must be **technically correct** when submitted digitally. A PDF with unembedded fonts may be garbled by applicant tracking systems. A PDF that is not accessible (no text layer, wrong reading order) fails accessibility requirements increasingly demanded by large employers. A DOCX generated from an over-specialised PDF workflow that loses structure when opened in Word is a practical failure.

The key typographic relationships in a CV:

- The candidate's name is the document's primary visual element. It should be notably larger than everything else — not decoratively large, but unambiguously the top of the hierarchy.
- Section headings create the major divisions. A horizontal rule under each section heading is the classic convention; colour is an alternative.
- Dates should be consistently right-aligned in the same column across all entries. The eye should be able to scan the date column to understand the timeline without reading the entry text.
- Bullet points should be genuine bullets — brief, parallel, beginning with active verbs — not sentences.

A single-column résumé: Markdown source, Typst PDF

The single-column format is the most widely applicable: appropriate for industry applications across almost every sector, readable as a printed document, and compatible with applicant tracking systems. It is also

the format best suited to a Markdown-first workflow, because the visual structure is simple and the same source can usually serve PDF, HTML, and DOCX without distortion.

The source should describe structure, not page mechanics. A minimal résumé source in Markdown:

```

---
name: "A. N. Author"
tagline: "Document Engineer & Typographer"
email: "author@example.edu"
phone: "+44 1234 567890"
website: "www.example.edu/~author"
---

## Experience

### Senior Document Engineer | Typeset Ltd | London | 2020–present

- Developed Pandoc pipeline producing PDF, HTML, and EPUB from one source
- Maintained internal publication templates and PDF preflight checks
- Reduced production time by 60%

### Technical Writer | Documentation Co. | Edinburgh | 2016–2020

- Wrote API documentation for three major software releases
- Introduced Git-based review workflow for documentation

```

That source is already the résumé’s logical structure: section headings, dated entries, and short achievement bullets. The PDF-specific work belongs in a Typst template:

```

#let accent = rgb("#1a4e8c")

#let cv-section(title) = {
  v(8pt, weak: true)
  text(fill: accent, weight: "bold", size: 12pt, title)
  line(length: 100%, stroke: 0.5pt + accent.lighten(40%))
  v(4pt, weak: true)
}

#let cv-entry(title, org, location, dates, body) = {

```

```

grid(
  columns: (1fr, auto),
  gutter: 6pt,
  [*#title*], text(size: 9pt, style: "italic", fill: luma(110), dates),
  text(style: "italic", org), text(size: 9pt, fill: luma(110),
↵ location),
)
body
v(5pt, weak: true)
}

#set page(paper: "a4", margin: (x: 25mm, y: 20mm))
#set text(font: "EB Garamond", size: 11pt)

#align(center)[
  #text(size: 22pt, weight: "bold")[A. N. Author]
  #v(3pt)
  #text(size: 9.5pt, fill: luma(105))[
    #link("mailto:author@example.edu")[author@example.edu]
    | +44 1234 567890 | www.example.edu/~author
  ]
]
#line(length: 100%, stroke: 1pt + accent)

```

The important point is the separation of concerns. Markdown owns the career data. Typst owns the page geometry, the right-aligned dates, the header rule, and the section styling. The template is short because Typst's grid and text primitives express the layout directly instead of forcing the author to build helper macros first.

A two-column layout

The two-column CV places primary content (experience, education) in a wide left column and secondary content (skills, contact information, languages) in a narrower right column — a layout common in design, technology, and creative fields. This is where a source-neutral workflow becomes harder: the design depends more heavily on page-specific composition, so Typst or LaTeX usually becomes the better PDF path.

In Typst, the implementation is explicit: a wide left column for the main narrative and a narrow right column for contact details, skills, and languages. The two columns should not be balanced automatically, because a CV's sidebar almost never has the same height as the main content.

```
#grid(
  columns: (2.1fr, 0.9fr),
  gutter: 14pt,
  [
    #cv-section("Experience")
    #cv-entry(
      "Senior Document Engineer",
      "Typeset Ltd",
      "London",
      "2020--present",
      [
        - Pandoc pipeline: PDF, HTML, EPUB from one source
        - Automated PDF preflight and font verification
      ],
    ),

    #cv-section("Education")
    *MSc Information Design*\
    University of Example, 2016
  ],
  [
    #block(
      inset: 10pt,
      fill: rgb("#f2f4f7"),
      radius: 4pt,
    ) [
      #cv-section("Skills")
      *Typesetting*\
      Typst, Pandoc, Quarto

      #cv-section("Languages")
      English, French
    ]
  ],
)
```

The sidebar's pale background is not decorative; it creates a second visual zone so the eye distinguishes supporting information from the main

employment narrative. The key compositional decision is top alignment: both columns begin together, then run independently.

The academic CV

An academic CV differs from an industry résumé in several important respects. It is longer — sometimes much longer, as it accumulates over a career. It has sections that do not appear on a short résumé: publications, grants, teaching, conference presentations, professional service, and sometimes administrative roles. The publications section in particular requires careful formatting, because publications are the primary evidence of academic productivity and must follow the citation conventions of the relevant field. A Markdown source remains valuable here, but the PDF backend often needs stronger bibliography support than a minimal résumé template.

For most academic CVs, keep the publication data in a bibliography database and render selected entries into the Markdown source. Pandoc's citation pipeline or Quarto's bibliography support is usually enough:

```
---
bibliography: publications.bib
csl: apa.csl
---

## Publications

### Journal Articles

1. @author2023typographic
2. @author2022reproducible

### Conference Papers

1. @author2024workflow
```

That structure is maintainable because the citation data lives in one place and the CV controls only grouping and order. If you need reverse chronology, put the newest key first in the Markdown list. If you

need a more complex publication taxonomy, keep one Markdown file per category and include them in the build.

The academic header should include institutional affiliation, office location, and ORCID in addition to the usual contact line:

```
---
name: "A. N. Author, MSc"
affiliation: "Department of Information Design, University of Example"
office: "Room 3.42, Springfield, SS 12345"
orcid: "https://orcid.org/0000-0000-0000-0001"
---
```

ORCID (Open Researcher and Contributor ID) is the standard persistent identifier for researchers. Linking it in the PDF or HTML output connects the CV to a maintained public record of publications and affiliations.

The primary workflow: single source, multiple outputs

A CV maintained as a LaTeX file is easy to produce as a PDF but awkward to send as an editable document (some employers request this), to publish as a web page, or to share in a format that non-LaTeX users can update. The better default is to maintain the content in Pandoc Markdown or Quarto and generate the final formats from that source.

The Markdown source uses the CV's natural structure:

```
---
name: "A. N. Author"
tagline: "Document Engineer & Typographer"
email: "author@example.edu"
phone: "+44 1234 567890"
website: "www.example.edu/~author"
---
```

Experience

Senior Document Engineer | Typeset Ltd | 2020–present

- Pandoc pipeline: PDF, HTML, EPUB from single source (60% time saving)
- Maintained house LaTeX document class (200+ publications)
- Automated PDF preflight and font verification

Education

MSc Information Design | University of Example | 2016

Distinction. Dissertation: Legibility of Serif Typefaces at Small Sizes

Skills

****Typesetting:**** LaTeX, Typst, Pandoc, groff
****Scripting:**** Python, Bash, Make
****Formats:**** PDF, HTML, EPUB, DOCX, man pages

For the PDF target, a small Typst template can read the same metadata and body content:

```
#import "@preview/cmmarker:0.1.5": render

#let resume(meta, body) = {
  set page(paper: "a4", margin: (x: 25mm, y: 20mm))
  set text(font: "EB Garamond", size: 11pt)

  align(center)[
    #text(size: 22pt, weight: "bold")[#meta.name]
    #if meta.tagline != none [
      #v(3pt)
      #emph[#meta.tagline]
    ]
    #v(3pt)
    #text(size: 9.5pt, fill: luma(105))[
      #meta.email | #meta.phone | #meta.website
    ]
  ]

  v(6pt, weak: true)
  line(length: 100%, stroke: 1pt + rgb("#1a4e8c"))
  v(6pt, weak: true)
  render(body)
}
```

The goal is the same as the Markdown source: keep the template thin. It should set page size, fonts, spacing, and header treatment once, then leave the content structure alone.

Render to all formats with a Makefile. In a modern workflow, the PDF target should prefer Typst when the design permits it and fall back to LaTeX only when the template requires LaTeX-specific layout control:

```
NAME := cv-author
SOURCE := cv-source.md
TMPL := styles/cv.typ
BUILD := build

.PHONY: all pdf html docx clean

all: pdf html docx

$(BUILD):
    mkdir -p $@

$(BUILD)/$(NAME).pdf: $(SOURCE) $(TMPL) | $(BUILD)
    pandoc $< --template=$(TMPL) --pdf-engine=typst -o $@

$(BUILD)/$(NAME).html: $(SOURCE) | $(BUILD)
    pandoc $< --standalone --css=cv.css -o $@

$(BUILD)/$(NAME).docx: $(SOURCE) | $(BUILD)
    pandoc $< -o $@

clean:
    rm -rf $(BUILD)
```

Running `make all` produces `build/cv-author.pdf`, `build/cv-author.html`, and `build/cv-author.docx` — three formats from a single source, rebuilt automatically when the source changes.

The HTML output benefits from a small stylesheet:

```
/* cv.css */
body {
    font-family: 'EB Garamond', Georgia, serif;
    font-size: 1.05rem;
```

```

line-height: 1.55;
color: #1c1c1c;
max-width: 800px;
margin: 2rem auto;
padding: 0 1.5rem;
}

h1 { font-size: 2rem; font-weight: 700;
      border-bottom: 2px solid #1a4e8c; padding-bottom: 0.3rem; }

h2 { font-size: 1.1rem; font-weight: 700; color: #1a4e8c;
      border-bottom: 1px solid #b0c4de; text-transform: uppercase;
      letter-spacing: 0.04em; margin: 1.5rem 0 0.5rem; }

h3 { font-size: 1rem; font-weight: 700; margin: 0.75rem 0 0.1rem; }

ul { margin: 0.3rem 0; padding-left: 1.5rem; }
li { margin-bottom: 0.15rem; }

@media print {
  body { max-width: none; margin: 0; padding: 0; font-size: 10pt; }
}

```

The DOCX output requires a reference document (`--reference-doc=cv-reference.docx`) to apply consistent styles. Create the reference document by opening the DOCX output in LibreOffice or Word, adjusting the heading and paragraph styles, and saving it as a template. On subsequent builds, `pandoc --reference-doc=cv-reference.docx cv-source.md -o cv.docx` applies those styles to the new content.

A limitation of the single-source approach is that complex two-column layouts, coloured sidebars, and publication-heavy academic headers are difficult to achieve through a thin Pandoc template alone. For those cases, keep the Markdown as the content source and give the PDF target a dedicated Typst layout. The single-source approach is most valuable for a straightforward professional résumé where the visual design is secondary to the content.

The Typst CV

For projects already using Typst, a CV template in Typst is clean and maintainable. In many cases it should be the preferred PDF path even when the source content begins in Markdown, because the helper functions that define entries and sections are written in Typst's functional style rather than LaTeX's macro language.

```
#let accent = rgb("#1a4e8c")

#let cv-section(title) = {
  v(8pt, weak: true)
  text(fill: accent, weight: "bold", size: 12pt, title)
  line(length: 100%, stroke: 0.5pt + accent.lighten(40%))
}

#let cv-entry(title, dates, org, location, details: none) = {
  grid(
    columns: (1fr, auto),
    [*#title*], text(style: "italic", size: 9pt, dates),
    text(style: "italic", org), text(size: 9pt, fill: luma(120),
↵ location),
  )
  if details != none { details }
  v(4pt, weak: true)
}

// Document setup
#set page(paper: "a4", margin: (x: 25mm, y: 20mm))
#set text(font: "EB Garamond", size: 11pt)

// Header
align(center)[
  #text(size: 22pt, weight: "bold")[A.#h(0.1em)N. Author]
  #v(3pt)
  #text(size: 9.5pt, fill: luma(100))[
    #link("mailto:author@example.edu")[author@example.edu]
    | +44 1234 567890 | www.example.edu
  ]
]
#line(length: 100%, stroke: 1.2pt + accent)

// Experience section
#cv-section("Experience")
```

```
#cv-entry(
  "Senior Document Engineer", "2020--present",
  "Typeset Ltd", "London",
  details: [
    - Pandoc pipeline: PDF, HTML, EPUB from single source
    - Maintained LaTeX class for 200+ publications
  ]
)
```

The `cv-entry` function uses a grid with `1fr` for the title column and `auto` for the dates column, which automatically right-aligns the dates to the line width — the same effect as `tabular*` with `\extracolsep{\fill}` in LaTeX, but expressed directly.

Compile:

```
typst compile cv.typ
```

Choosing the approach

For an industry résumé (one or two pages, clean professional appearance): keep the source in Markdown and generate PDF, HTML, and DOCX from it. Prefer Typst for PDF if you want a proper print layout without giving up the shared source.

For a designed two-column résumé: keep the content in Markdown if possible, but expect the PDF output to move into a dedicated Typst layout. That is the right place to solve columns, sidebars, and spacing.

For an academic CV with a publications section: keep the data source structured and reproducible. Put the publication data in a bibliography file, group entries in Markdown, and render the final PDF through Typst or Pandoc according to the complexity of the design.

For legacy workflows: use LaTeX only when a department, journal, or inherited template stack leaves no realistic alternative.

The résumé is a document that will be revised regularly throughout a career. Whatever approach you choose, it should be one you can maintain without friction — one where updating a job title or adding a new publication is a matter of editing a line, not navigating a complex template structure.

Articles and Reports

The article and the report are the workhorses of technical and academic publishing. Between them, they cover the range from a four-page conference paper to a two-hundred-page internal report, from a single-author journal submission to a multi-author technical specification. The typographic requirements of these documents are well understood — decades of published guidelines and house styles have codified what a professional article looks like — which makes them ideal territory for demonstrating how CLI tools handle the full complexity of real documents.

This chapter builds three examples from a Markdown-first source tree. The first is a journal-style article with an abstract, numbered sections, mathematical equations, a table, cross-references, and a bibliography. The second is a technical report with a title page, table of contents, list of figures, code listings, appendices, and a two-sided layout. The third is a two-column magazine-style layout where a Typst or Pandoc PDF path is the first choice and LaTeX's `twocolumn` mode remains a fallback when a legacy class or package stack is required.

A journal-style article

The default workflow for an article should begin with Markdown plus metadata, not with a handwritten TeX preamble. That keeps HTML, DOCX, EPUB, and PDF available from one source, keeps the manuscript readable in Git, and makes Quarto or Pandoc the controlling layer rather than the PDF backend. Typst is the natural PDF-first companion for this workflow: it handles page layout, maths, figures, and bibliography cleanly without forcing the author to write TeX macros. LaTeX still matters when a journal mandates a class file or when a specific package ecosystem is unavoidable.

When a LaTeX backend is required, the article typically uses the `article` class, occasionally a publisher-supplied class, with a standard set of packages for typography, mathematics, bibliographies, and cross-references.

Document structure and metadata

For a Markdown-first article, the equivalent of the old preamble is the metadata block plus a small PDF template. The metadata should describe structure, bibliographic resources, and numbering rules:

```

---
title: "On the Typographic Quality of CLI-Produced Documents"
subtitle: "A Systematic Comparison"
author:
  - name: "A. N. Author"
    affiliation: "Department of Information Design, University of Example"
    email: "author@example.edu"
  - name: "B. M. Collaborator"
    affiliation: "Institute of Typography, Townsley University"
date: "2024-03-15"
abstract: |
  This paper presents a systematic comparison of documents produced
  using CLI-based typesetting tools against those produced using
  graphical desktop publishing applications.
keywords: [typography, CLI, Pandoc, Typst, reproducibility]
bibliography: references.bib
numbersections: true
---

```

That is enough information for Pandoc or Quarto to generate HTML, EPUB, or PDF. The PDF-specific styling then lives in Typst, where page geometry, heading style, theorem blocks, and figure captions can be expressed directly.

```

#set page(paper: "a4", margin: (x: 25mm, y: 25mm))
#set text(font: "Libertinus Serif", size: 11pt)
#set par(justify: true)

#show heading.where(level: 1): it => block(above: 1.4em, below: 0.7em)[

```

```

#set text(weight: "bold", size: 18pt)
#it
]

#let theorem(kind, title, body) = block(
  inset: 10pt,
  radius: 4pt,
  stroke: 0.4pt + rgb("#9aa7b2"),
  fill: rgb("#f8fafc"),
)[
  *#kind.* #title
  #v(4pt)
  #body
]

```

Title and authorship

The title block belongs in metadata, not in backend markup. When you need a more formal multi-author rendering, keep the data structured and let the template format it:

```

author:
- name: "A. N. Author"
  affiliation:
    - "Department of Information Design, University of Example"
  email: "author@example.edu"
  corresponding: true
- name: "B. M. Collaborator"
  affiliation:
    - "Institute of Typography, Townsley University"

```

In Typst, the title block can be a small helper function that formats names prominently and affiliations secondarily:

```

#let article-title(meta) = align(center)[
  #text(size: 20pt, weight: "bold")[#meta.title]
  #if meta.subtitle != none [
    #v(5pt)
    #text(size: 12pt, style: "italic")[#meta.subtitle]
  ]
]

```

```

#v(10pt)
#for person in meta.author [
  *#person.name*\
  #for line in person.affiliation [#line\ ]
  #if person.email != none [#link("mailto:" +
  ↪ person.email)[#person.email]]
  #v(8pt)
]
]

```

Abstract and keywords

Abstracts and keywords are best expressed directly in the front matter and rendered consistently by the article template. In Markdown:

```

abstract: |
  This paper presents a systematic comparison of documents produced
  using CLI-based typesetting tools against those produced using
  graphical desktop publishing applications. Our principal finding is
  that CLI-produced documents match or exceed the quality of graphical
  alternatives when the workflow is configured deliberately.
keywords:
- typography
- CLI tools
- Pandoc
- Typst
- reproducibility

```

The template can then decide whether the abstract is centred, narrower than the body text, or separated by a rule. That keeps the source portable while preserving print discipline.

Mathematics

Keep mathematics in the Markdown body so it survives every output target. Quarto-style labels or pandoc-crossref labels make equations referenceable without handwritten backend commands:

Following [bringhurst2012], we define quality as:

```
$$
Q = \sum_{i=1}^5 w_i q_i
$$ {#eq-quality}
```

where q_i are the individual dimension scores and w_i are the weights derived from expert assessment. Equation @eq-quality shows that overall quality is a weighted linear combination.

That same equation can be rendered by HTML math support, Typst, or a LaTeX fallback without changing the source.

Tables

Simple article tables should begin as Markdown, because the same source then works in HTML, EPUB, and PDF:

Category	Count	Mean pages
Academic articles	42	14.3
Technical reports	35	28.7
Correspondence	28	2.1
Reference manuals	15	67.4
Total	120	19.8

: Document corpus composition {#tbl-corpus}

If the print layout needs stronger control, move the table styling into Typst rather than rewriting the data in a different source language:

```
#table(
  columns: (2fr, auto, auto),
  stroke: none,
  table.header(
    [*Category*], [*Count*], [*Mean pages*],
  ),
  [Academic articles], [42], [14.3],
  [Technical reports], [35], [28.7],
```

```
[Correspondence], [28], [2.1],
[Reference manuals], [15], [67.4],
table.hline(),
[*Total*], [*120*], [*19.8*],
)
```

Cross-references

Cross-references should live at the source level, not in backend commands. With `Quarto` or `pandoc-crossref`, the body text stays readable:

```
As shown in @tbl-corpus, our dataset contains 120 documents.
The quality equation (@eq-quality) defines the scoring metric.
See also @fig-results for the final comparison.
```

This is the right habit for every backend. Hard-coded numbers break as soon as the document is reorganised.

Bibliography

Bibliographic data should stay in a citation database and be cited from the Markdown source:

```
Typography was defined by [@bringhurst2012] as the craft of endowing
human language with a durable visual form.
```

```
The line-breaking algorithm discussed in [@knuth1984] considers the
paragraph as a whole rather than individual lines.
```

In a direct Typst workflow, the bibliography can be attached explicitly:

```
#bibliography("references.bib", style: "apa")
```

In a Pandoc or `Quarto` workflow, `--ci` `teproc` or `Quarto`'s built-in citation processing resolves the same keys for HTML, EPUB, and PDF.

The primary Pandoc or Quarto article workflow

For articles that need multiple output formats — HTML for a preprint server, PDF for review or print, DOCX for a co-author who uses Word — Pandoc Markdown or Quarto should be the canonical source. Use Typst for PDF when you want a programmable print layout with less template overhead; keep a LaTeX template only for venues that depend on LaTeX packages or class files.

The Markdown front matter replicates the LaTeX title block:

```

---
title: "On the Typographic Quality of CLI-Produced Documents"
subtitle: "A Systematic Comparison"
author:
  - "A. N. Author, University of Example"
  - "B. M. Collaborator, Townsley University"
date: "15 March 2024"
abstract: |
  This paper presents a systematic comparison of documents produced
  using CLI-based typesetting tools against those produced using
  graphical desktop publishing applications. Our principal finding is
  that CLI-produced documents, when configured appropriately, match or
  exceed the quality of graphical alternatives.
keywords: [typography, CLI, Pandoc, Typst, reproducibility]
bibliography: references.bib
numbersections: true
---
```

The equivalent Typst template can stay much thinner because the layout logic is built into the language rather than spread across a preamble:

```

#let article(meta, body) = {
  set page(paper: "a4", margin: (x: 25mm, y: 25mm))
  set text(font: "Libertinus Serif", size: 11pt)
  set par(justify: true)

  article-title(meta)

  if meta.abstract != none [
    v(10pt, weak: true)
  ]
}
```

```

block(inset: 10pt, fill: rgb("#f8f9fc"))[
  *Abstract.* #meta.abstract
  if meta.keywords != none [
    #v(5pt)
    *Keywords:* #meta.keywords.join(", ")
  ]
]

v(12pt, weak: true)
body
bibliography("references.bib")
}

```

The important design principle is that the template consumes metadata and body content without forcing the author to duplicate structure in the PDF layer.

Render to HTML and a default Typst-backed PDF:

```

pandoc article.md --standalone --toc --number-sections \
  --citeproc -o article.html

pandoc article.md --citeproc --number-sections \
  --pdf-engine=typst -o article.pdf

```

If a venue requires a legacy PDF backend, switch only the PDF command and keep the Markdown source unchanged.

For cross-references within the Pandoc workflow, the `pandoc-crossref` filter adds `@fig-label`, `@tbl-label`, and `@eq-label` syntax analogous to Quarto's built-in cross-references:

```

pandoc article.md --filter=pandoc-crossref --citeproc -o article.pdf

```

With `pandoc-crossref` installed, references in Markdown become:

```
As shown in @tbl-corpus, our dataset contains 120 documents.  
The quality equation (@eq-quality) defines our metric.
```

which produces “as shown in Table 1” and “the quality equation (Equation 1)” in the output.

A technical report

A technical report differs from a journal article in scale and structure: it typically has a formal title page, a table of contents, a list of figures and tables, multiple chapters, code listings, appendices, and a two-sided layout for printing. The source should still stay in Markdown chapters plus defaults files. For most house styles, Typst is a better PDF-first target than a large report preamble; a native LaTeX report backend is mainly justified when an institution already owns a class file or requires specific packages.

Document structure

For reports, keep the source split into Markdown chapters and let the PDF backend add the title page, table of contents, appendices, and running heads. A typical project:

```
report/  
  □□□ report.qmd  
  □□□ sections/  
    □ □□□ 01-summary.md  
    □ □□□ 02-method.md  
    □ □□□ 03-results.md  
  □□□ references.bib  
  □□□ styles/  
    □□□ report.typ  
    □□□ report.css
```

The source files remain content-first. Page geometry, front matter, and print-specific rules live in `report.typ`.

Title page

The report's front matter can be described in metadata and formatted in Typst:

```

---
title: "CLI-Based Document Production"
subtitle: "Automation and Quality in Practice"
report-number: "TR-2024-03"
author:
  - "A. N. Author"
  - "B. M. Collaborator"
institution: "Department of Information Design, University of Example"
date: "March 2024"
summary: |
  This report describes the design and evaluation of an automated
  document production system that reduces production time by 60%
  while maintaining typographic quality.
---

```

```

#align(center)[
  #text(size: 14pt, weight: "bold")[TR-2024-03]
  #v(16pt)
  #text(size: 24pt, weight: "bold")[CLI-Based Document Production]
  #v(6pt)
  #text(size: 13pt, style: "italic")[Automation and Quality in Practice]
  #v(18pt)
  A. N. Author and B. M. Collaborator\
  Department of Information Design\
  University of Example
  #v(18pt)
  March 2024
]

```

Front matter

The report template should generate the standard prefatory material automatically:

```
#outline(title: [Contents])
#pagebreak()
#outline(title: [Figures], target: figure.where(kind: image))
#pagebreak()
#outline(title: [Tables], target: figure.where(kind: table))
```

These lists are optional in short reports but expected in formal, print-oriented work.

Code listings

Code listings should remain ordinary fenced code blocks in the Markdown source:

```
```yaml
#| label: lst-config
#| tbl-cap: Production defaults file
from: markdown
to: pdf
pdf-engine: typst
mainfont: "EB Garamond"
toc: true
```
```

Quarto and Pandoc can style those listings for HTML automatically, while the Typst template controls code font, background, numbering, and caption spacing for PDF output.

Subfigures

Related figures can be grouped in source with a layout div or a Typst grid:

```
#figure(
  caption: [Line-breaking quality before and after optimisation],
  grid(
    columns: 2,
```

```

gutter: 10pt,
figure(image("fig-before.png"), caption: [Before optimisation]),
figure(image("fig-after.png"), caption: [After optimisation]),
),
) <fig-comparison>

```

In the text, refer to @fig-comparison from Markdown or Quarto source.

Appendices

Appendices should be ordinary Markdown chapters flagged as appendices in the build order:

```

# Appendix: Installation Guide {#app-install}

## Prerequisites

...

```

If the PDF template needs lettered appendices, that numbering rule belongs in the template, not in the chapter source.

Running headers for reports

Running headers are another backend concern. In Typst, the page setup can pull the current heading into the header area:

```

#set page(
  header: context {
    let here = counter(heading).get()
    align(center)[#smallcaps[Section #here.at(0)]]
  },
  footer: context align(center)[#counter(page).display()],
)

```

The principle is the same whatever backend you use: the content source should not carry page furniture.

A two-column magazine layout

The two-column layout is the standard for many academic journals and some technical magazines. In a Markdown-first workflow, this is usually the point where Typst becomes more attractive than raw LaTeX: columns, placed figures, and headline styling can live in a PDF template instead of in an expanding preamble. LaTeX's `twocolumn` option is still worth understanding because some institutional workflows still depend on it, but it should be treated as a compatibility path rather than the default starting point.

The `twocolumn` class option

For a modern two-column article, it is usually cleaner to define the columns in Typst or CSS rather than rely on an old document-class switch. In Typst:

```
#set page(paper: "a4", margin: (x: 15mm, y: 20mm))
#set text(font: "Source Serif 4", size: 10pt)

#set par(justify: true)
#show heading.where(level: 1): it => block(columns: 2, above: 1em, below:
  ↪ 0.6em)[
  #set text(size: 18pt, weight: "bold")
  #it
]
```

The column gap is a design choice, not a source-level concern. On A4 with 10-point type, six to eight millimetres is a useful starting range.

A full-width title block

Two-column layouts still need a full-width title block and abstract. In Typst, place those before the body columns begin:

```
#align(center)[
  #text(size: 22pt, weight: "bold", fill: rgb("#12436d"))[
    Typography at the Command Line
  ]
  #v(5pt)
  #text(size: 12pt, style: "italic")[
    Why serious document production belongs in a terminal
  ]
  #v(8pt)
  #line(length: 100%, stroke: 0.6pt + rgb("#12436d"))
]

#block(inset: (x: 8mm, y: 5mm), fill: rgb("#f8fafc"))[
  *Abstract.* The abstract spans the full page width before the
  two-column body begins.
]
```

Wide figures and tables in two-column mode

Wide figures should be treated as explicit layout events. Put them between columned sections rather than forcing the engine to guess:

```
#figure(
  image("wide-figure.png", width: 90%),
  caption: [A wide figure spanning both columns],
) <fig-wide>
```

In HTML output, the same effect is often better achieved with CSS grid and a figure that spans all columns.

Column balance

The last page of a two-column article should not end with one long column and one stub unless there is a strong editorial reason. In practice, balance is handled by editing: cut or expand the text slightly, move a figure, or move a note to the following page. That is easier to reason about in a Typst or CSS layout than in a float-heavy legacy workflow.

Drop caps

A drop capital at the start of the first paragraph is a classic magazine convention. In HTML it is a CSS concern:

```
.lede::first-letter {  
  float: left;  
  font-size: 3.2em;  
  line-height: 0.9;  
  padding-right: 0.08em;  
  font-family: "Source Serif 4", serif;  
}
```

For PDF, the same treatment belongs in the Typst template or style module, not in the article's source text.

Building articles from Markdown with Typst or LaTeX

The same Markdown source can drive both the journal-style and the two-column outputs with different PDF backends. In a modern workflow, keep Typst as the default PDF target and swap away only when a legacy class or package requirement forces the change.

```
# article-defaults.yaml  
from: markdown  
to: pdf  
pdf-engine: typst  
template: styles/article.typ  
citeproc: true  
number-sections: true  
toc: false
```

```
# twocol-defaults.yaml  
from: markdown  
to: pdf  
pdf-engine: typst  
template: styles/twocol.typ  
citeproc: true  
number-sections: false
```

The two-column template changes only the page geometry, title treatment, and article body layout. The source content remains identical.

For the Makefile:

```
article.pdf: article.md styles/article.typ references.bib
    pandoc --defaults=article-defaults.yaml $< -o $@

twocol.pdf: article.md styles/twocol.typ references.bib
    pandoc --defaults=twocol-defaults.yaml $< -o $@

article.html: article.md
    pandoc --standalone --citeproc --number-sections $< -o $@
```

The same `article.md` produces three different outputs: a single-column PDF, a two-column PDF, and an HTML version — all from one source, all rebuilt automatically when the content changes.

The three document types in this chapter — journal article, technical report, two-column layout — cover the range of what academic and technical writers produce most often. The patterns established here recur in Chapter 22 (Books), where the same approaches scale up to book-length projects with chapters, an index, and a complete front and back matter apparatus.

Presentations

A presentation is a constrained document: a fixed sequence of screens, each carrying a limited amount of information, structured to be read not by a reader at their own pace but by an audience following a speaker. The typographic requirements are different from those of a printed document — larger type, stronger contrast, simpler layouts — and the relationship between source and output is different too: a presentation must work as a PDF for in-person delivery, as an HTML file for online sharing, and sometimes as a handout that compresses multiple slides onto a printed page.

For most presentation work, the default CLI path should start with Markdown and target Reveal.js or Quarto's `revealjs` format. That keeps the deck readable in source control, easy to publish on the web, and simple to revise quickly. A PDF deck is still useful for venues that require offline delivery, strict archival formats, or print handouts, but it should be treated as a secondary target produced from the same source rather than as the workflow's centre of gravity. This chapter therefore treats Markdown and Reveal.js as the primary path, with Quarto as the executable extension and PDF export as a derivative step.

Reveal.js as the default deck

Reveal.js is the right default because the source stays plain Markdown, the delivered artifact is a web-native deck, and speaker notes, slide navigation, and print styles are already built into the presentation layer.

Metadata and themes

```

---
title: "CLI Typography"
subtitle: "From Markdown to Slides"
author: "A. N. Author"
date: "March 2024"
format:
  revealjs:
    theme: white
    slide-number: true
    transition: fade
    width: 1600
    height: 900
    chalkboard: false
---

```

Most modern projection uses 16:9. Whether you set that explicitly with pixel dimensions or inherit it from the theme, keep the deck designed for widescreen unless the venue says otherwise.

Slides, columns, and callouts

Every slide should be a heading plus a small amount of structured content:

```

## The CLI Advantage

CLI tools separate content from presentation.

- Source in Markdown = version-controllable
- Multiple outputs from one source
- Reproducible: same input, same output, always

```

Side-by-side slides are expressed with Pandoc or Quarto fenced divs rather than backend-specific frame syntax:

```

## Side-by-Side Columns

::::: {.columns}
::: {.column width="50%"}
**Print**

- Stable PDF export
- Predictable handouts
- Good for archives
:::
::: {.column width="50%"}
**Web**

- Speaker notes
- Browser delivery
- Easy sharing
:::
:::::

```

Incremental reveals

Incremental reveals should be authored in Markdown, not in slide-engine commands:

```

## Building up content

First block appears immediately.

. . .

This block appears after the first pause.

. . .

And this after the second.

```

Use reveals sparingly. They are useful when they control pacing; they are noise when they merely dramatise a static list.

Speaker notes

Speaker notes belong in the source as fenced divs:

```
## Key findings

- CLI tools match desktop publishing quality with appropriate setup
- Reproducibility is the main operational advantage

::: notes
Emphasise the "appropriate setup" qualifier. The argument is not that
defaults are perfect, but that the workflow is controllable.
:::
```

That works in Reveal.js speaker view and in Quarto-generated decks without turning the slide source into presenter markup.

Handouts and PDF export

When you need a PDF handout, export it from the same HTML deck rather than rebuilding the deck in a different language. Reveal.js supports print styles, and tools such as dektape can capture the rendered presentation:

```
pandoc slides.md -t revealjs --standalone \
  -V theme=white \
  -V slideNumber=true \
  -V transition=fade \
  -o slides.html

dektape reveal slides.html slides.pdf
```

For a handout, either use a custom print stylesheet or generate a second HTML build with overlays disabled and smaller margins.

Markdown source shared across slide targets

Pandoc converts the same Markdown source to multiple slide targets. In practice, that usually means Reveal.js HTML as the primary artifact and an exported PDF as the secondary artifact. Level-2 headings (##) become slide titles, level-1 headings (#) become section dividers when the theme supports them, and horizontal rules can also force breaks.

A minimal slide deck in Pandoc Markdown:

```
---
title: "CLI Typography"
subtitle: "From Markdown to Slides"
author: "A. N. Author"
institute: "University of Example"
date: "March 2024"
theme: metropolis
transition: fade
slideNumber: true
---

# The Problem

## GUI Slides

- Inconsistent formatting across slides
- Not version-controllable
- Hard to maintain consistency

## Side-by-Side Columns

::::::::::: {.columns}
::: {.column width="50%"}
**PDF Export**

- Exported handout
- Useful for archives
- Venue fallback
:::
::: {.column width="50%"}
**Reveal.js (HTML)**

- Interactive
- Speaker notes
```

```
- Runs in browser
:::
:::~::~:
```

Compile the shared Markdown source to HTML first:

```
pandoc slides.md -t revealjs --standalone \
  -V theme=white \
  -V slideNumber=true \
  -o slides.html
```

Incremental reveals in Pandoc Markdown use . . . (three dots on their own line) to insert a \pause:

```
## Building up content

First block appears immediately.

. . .

This block appears after the first pause.

. . .

And this after the second.
```

Speaker notes use a fenced div with class notes:

```
## Key findings

- CLI tools match print-quality output with appropriate configuration

::: notes
Emphasise the "appropriate configuration" qualifier. The point is
that out-of-the-box defaults are not enough; intentional setup is
↪ required.
:::
```

Key presentation metadata variables for the YAML front matter:

| Variable | Effect |
|---------------|--|
| theme | Visual theme (white, black, league, etc.) |
| transition | Transition style (fade, slide, zoom) |
| slideNumber | Show slide numbers |
| width, height | Slide dimensions |
| slide-level | Heading level creating slides (default: 2) |
| controls | Show navigation controls |
| progress | Show progress bar |
| center | Vertically centre slide content |

For a handout build, switch to a print stylesheet or export the Reveal.js HTML to PDF after rendering.

Reveal.js: HTML presentations

Reveal.js produces interactive HTML presentations. The same Markdown source compiles directly to an HTML deck:

```
pandoc slides.md -t revealjs --standalone \
  -V theme=white \
  -V slideNumber=true \
  -V transition=fade \
  -o slides.html
```

The output is a self-describing HTML file that loads Reveal.js from a CDN (or a local copy). Open it in a browser to present; the `f` key enters full-screen, `s` opens the speaker notes view, `b` blacks out the screen.

Reveal.js themes available through Pandoc: black, white, league, beige, sky, night, serif, simple, solarized, blood, moon.

Transition styles for slide transitions: none, fade, slide, convex, concave, zoom.

Key Reveal.js metadata variables:

| Variable | Effect |
|---------------|--|
| theme | Visual theme |
| transition | Slide transition style |
| slideNumber | Show slide numbers (true/false) |
| controls | Show navigation controls |
| progress | Show progress bar |
| history | Enable browser history navigation |
| center | Vertically centre slide content |
| width, height | Slide dimensions in pixels |
| margin | Slide margin as fraction of screen |
| revealjs-url | Path to Reveal.js files (for local installation) |

For a standalone file that can be distributed or archived without an internet connection, install Reveal.js locally and reference it:

```
npm install reveal.js

pandoc slides.md -t revealjs --standalone \
  -V revealjs-url=./node_modules/reveal.js \
  -o slides.html
```

The `revealjs-url` variable overrides the default CDN path. With a local Reveal.js installation, the HTML file is fully self-contained within its directory and works without internet access.

Speaker notes in Reveal.js are visible in the speaker view (s key) but not on the main display. They use the same `:::` notes fenced div syntax shown earlier.

The **slide-level** option controls which heading level creates slides. With `--slide-level=1`, level-1 headings (#) create slides and level-2 headings (##) create subsection breaks within slides. With `--slide-level=2` (the default), level-1 headings create section title slides and level-2 headings create content slides. Choose based on how your source is structured.

Producing from a common source

The value of the Pandoc approach is producing both a browser-native deck and a PDF derivative from the same Markdown source. In most workflows, the HTML deck is the primary artifact and the PDF is the fallback or handout. A Makefile handles this:

```
SLIDES := slides
BUILD := build

.PHONY: all pdf html clean

all: html pdf

$(BUILD):
    mkdir -p $@

$(BUILD)/$(SLIDES).html: $(SLIDES).md | $(BUILD)
    pandoc $< -t revealjs \
        --standalone \
        -V theme=white \
        -V slideNumber=true \
        -V transition=fade \
        -o $@

$(BUILD)/$(SLIDES).pdf: $(BUILD)/$(SLIDES).html
    decktape reveal $< $@

clean:
    rm -rf $(BUILD)
```

Running `make all` produces `slides.html` and `slides.pdf` from the same Markdown source. Structural changes (adding a slide, revising a bullet point, adding a new section) are made once and propagate to both outputs.

Quarto for slides with executable code

When presentation slides contain generated figures — data visualisations, simulation results, plots from an analysis — Quarto provides code exe-

cution in presentation format. Covered in Chapter 14, the key points for presentations specifically:

A Quarto presentation uses the same `.qmd` format as other Quarto documents, with `revealjs` as the default output format:

```
---  
format:  
  revealjs:  
    theme: default  
    slide-number: true  
---
```

Code blocks in slides execute and embed their output directly:

```
#| echo: false  
#| fig-width: 6  
#| fig-height: 3.5  
  
import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.linspace(0, 2*np.pi, 100)  
fig, ax = plt.subplots()  
ax.plot(x, np.sin(x))  
ax.set_xlabel("x")  
plt.tight_layout()  
plt.show()
```

The figure appears on the slide without any manual export or inclusion step. When the analysis changes, recompiling the presentation rebuilds the figure automatically.

For presentations in scientific and academic contexts where results must be current and reproducible, this is the right approach. For presentations that do not involve computation, plain Pandoc Markdown with Reveal.js is simpler and has fewer dependencies.

Presentations complete the standard range of document types for professional and academic work. The important pattern is the same as elsewhere in the book: write in Markdown, publish to the web by default, and generate PDF only when the context requires it. The next chapter — Books — scales that pattern up to the most complex document type: a work with dozens of chapters, a complete front matter apparatus, an index, and output in multiple formats simultaneously.

Books — The Complete Project

A book is the most complex document type a typographer works with. It is not simply a long article. It has a front matter apparatus — half title, title page, copyright notice, dedication, table of contents, list of figures, preface — and a back matter apparatus — bibliography, index, colophon — that frame the main text. It is structured as a hierarchy of parts, chapters, and sections. It has running headers that reflect the current chapter and section. It may have an index with thousands of entries cross-referenced to specific pages. Its bibliography may contain hundreds of sources formatted to a specific style. And all of this must work simultaneously in PDF, EPUB, and HTML, each format imposing different constraints.

This chapter builds the complete book production pipeline from Markdown chapters, shared metadata, and reproducible build commands. The default path is multi-format: HTML and EPUB from the same sources, plus PDF through a dedicated print backend such as Typst. A native LaTeX book remains useful for print-only projects or publisher-mandated workflows, but it should be the exception rather than the organising principle. The chapter therefore treats the Markdown source tree as canonical and the PDF backend as a replaceable implementation detail.

Project structure

A book project's directory structure should reflect its logical organisation:

```
book/
```

```

□□□ Makefile                ← build system
□□□ metadata.yaml          ← document-wide metadata
□□□ references.bib         ← bibliography database
□□□ chapters/
□   □□□ 00-preface.md
□   □□□ 01-history.md
□   □□□ 02-fundamentals.md
□   □□□ ...
□□□ styles/
□   □□□ book.typ           ← Typst template for PDF
□   □□□ book-template.latex ← LaTeX fallback template
□   □□□ web.css           ← CSS for HTML output
□   □□□ epub.css         ← CSS for EPUB output
□□□ assets/
□   □□□ cover.jpg
□   □□□ figures/
□       □□□ pipeline.pdf
□       □□□ pipeline.svg
□□□ build/                ← generated files (in .gitignore)
□   □□□ book.pdf
□   □□□ book.html
□   □□□ book.epub
□□□ .gitignore

```

The `chapters/` directory contains one Markdown file per chapter, numbered with leading zeros to ensure correct alphabetical sort order. The `metadata.yaml` file holds the title, author, bibliography settings, and format-specific options that apply to the entire book. The PDF template can be Typst or LaTeX; the source tree stays the same either way. The `build/` directory is generated and is excluded from version control.

`.gitignore` for the book project:

```

build/
*.aux
*.log
*.toc

```

```
*.out  
*.lof  
*.lot  
*.idx  
*.ind  
*.ilg  
*.bbl  
*.blg  
*.fls  
*.fdb_latexmk  
*.synctex.gz
```

The PDF backend: Typst first, LaTeX when needed

For a Markdown-first book, the PDF layer should be swappable. Typst is the cleaner default when you want a programmable print layout without inheriting TeX's macro language. The book template should own geometry, running heads, chapter openings, and front and back matter conventions.

```
#set page(  
  paper: "a4",  
  margin: (  
    inside: 30mm,  
    outside: 25mm,  
    top: 30mm,  
    bottom: 25mm,  
  ),  
)  
#set text(font: "EB Garamond", size: 11pt, lang: "en")  
#set par(justify: true)
```

The inside and outside margins matter in the same way that gutter and fore-edge margins matter in traditional book work. The source files should not know about any of this; they should remain chapter files.

Front matter

The book begins with front matter — content before the first main chapter, typically paginated separately and laid out more quietly than the main text. In a Typst template, this is usually one function per front-matter element:

```
#let title-page(meta) = pagebreak(to: "odd") + align(center)[
  #v(35%)
  #text(size: 26pt, weight: "bold")[#meta.title]
  #v(8pt)
  #text(size: 14pt)[#meta.subtitle]
  #v(18pt)
  #meta.author
  #v(24pt)
  #meta.publisher
]

#let copyright-page(meta) = pagebreak(to: "odd") + block[
  Copyright © #meta.year #meta.author.
  All rights reserved.
]

#let dedication(text) = pagebreak(to: "odd") + align(center + horizon)[
  #emph[text]
]

#outline(title: [Contents])
```

The point is not that these exact helper functions are universal. The point is that front matter is a template responsibility, not something the author should retype in every manuscript.

Main matter and parts

```
# Part: Foundations

# A Brief History of Typesetting {#ch-history}
```

```
Body text with index markers and citations.
```

```
## Hot Metal to Digital
```

```
More text...
```

```
# Part: The Toolbox
```

```
# Pandoc: The Universal Converter
```

The source should reflect the book’s logical outline only. Whether part openings begin on recto pages, whether chapter titles occupy a full spread, and how running heads are updated are all template decisions.

Back matter

```
# Bibliography
```

```
# Index
```

```
# Colophon
```

In the source tree, these can be generated sections or included back-matter files. The colophon — a brief note about how the book was produced — appears on the final page. It is the natural place to record the toolchain, typefaces, and workflow.

Running headers

Running headers display context-sensitive information as the reader navigates the book. The standard book convention:

- **Left (verso) pages:** chapter title in small italic, page number at outer edge

- **Right (recto) pages:** section title in small italic, page number at outer edge

```
#set page(
  header: context {
    let loc = here()
    align(center)[#smallcaps[Chapter] #query(heading.where(level:
↵ 1)).last().body]
  },
  footer: context align(center)[#counter(page).display()],
)
```

The details vary by template, but the principle is stable: running heads belong to the page layer, not the chapter source. Chapter-opening pages can use a separate page style with the header suppressed.

Index generation

An index is still a multi-step process, but the source should remain Markdown. The usual pattern is to mark terms in the source and let a filter or generator collect them into an index file.

Marking index entries

Mark indexable terms semantically:

```
Typography [typography]{.index} is the craft of endowing language
with a durable visual form.
```

```
The baseline [baseline]{.index} is the invisible line on which
letters rest.
```

For subentries or see-also references, use attributes:

```
[serif]{.index main="typeface" sub="serif"}
[LaTeX]{.index see="TeX"}
```

Running makeindex

After the main render step, run the index generator on the collected terms:

```
pandoc metadata.yaml chapters/*.md \  
  --lua-filter=filters/index.lua \  
  -o build/book-index.json  
  
typst compile styles/book.typ build/book.pdf
```

The exact mechanics depend on the indexing tool you choose, but the goal is the same as everywhere else in this book: keep the manuscript human-readable and let the build step assemble the machinery.

xindy as an alternative

If the book is multilingual, use an indexing tool that understands locale-aware sorting and Unicode. The important requirement is not the historical tool name but correct alphabetical order in the finished index.

Bibliography

For a Markdown-first book, bibliography data should stay in `references.bib` or another citation database, and the manuscript should cite keys directly:

```
Typography was defined by [bringhurst2012] as the craft of endowing  
human language with a durable visual form.
```

```
The line-breaking algorithm discussed in [knuth1984] considers the  
paragraph as a whole rather than individual lines.
```

In Typst, the bibliography is attached at the template level:

```
#bibliography("references.bib", style: "chicago-author-date")
```

In Pandoc or Quarto, `--citeproc` performs the same job for HTML, EPUB, and PDF.

The Pandoc multi-format build

For books that need simultaneous PDF, HTML, and EPUB output, Pandoc Markdown provides the source-neutral authoring path. Each chapter is a separate `.md` file; the book's structure and metadata live in `metadata.yaml`. This is the path that should dominate the workflow. The PDF backend can be Typst by default and changed later if a print requirement forces a LaTeX-specific template.

```
# metadata.yaml
title: "The CLI Typographer"
subtitle: "Typography, Typesetting, and Document Production from the
↳ Command Line"
author: "A. N. Author"
date: "2024"
lang: en-GB
documentclass: book
mainfont: "EB Garamond"
sansfont: "Source Sans 3"
monofont: "JetBrains Mono"
toc: true
toc-depth: 2
numbersections: true
bibliography: references.bib
```

Chapter files contain only their content, with the chapter heading as the first heading:

```
# A Brief History of Typesetting

Typography is the craft of endowing human language with a durable
visual form [bringhurst2012].
```

```
## Hot Metal to Digital
```

The transition began in the 1950s...

Build all three formats:

```
# PDF via Typst
pandoc --citeproc \
  --bibliography=references.bib \
  --pdf-engine=typst \
  --toc --toc-depth=2 \
  --number-sections \
  metadata.yaml chapters/*.md \
  -o build/book.pdf

# HTML
pandoc --citeproc \
  --bibliography=references.bib \
  --standalone --toc --toc-depth=2 \
  --number-sections \
  --css=styles/web.css \
  metadata.yaml chapters/*.md \
  -o build/book.html

# EPUB
pandoc --citeproc \
  --bibliography=references.bib \
  --split-level=1 --toc \
  --epub-cover-image=assets/cover.jpg \
  --css=styles/epub.css \
  metadata.yaml chapters/*.md \
  -o build/book.epub
```

A complete Makefile for this pipeline:

```
BOOK      := cli-typographer
CHAPTERS  := $(sort $(wildcard chapters/*.md))
META      := metadata.yaml
BIB       := references.bib
BUILD     := build
PANDOC    := pandoc
```

```

.PHONY: all pdf html epub clean

all: pdf html epub

$(BUILD):
    mkdir -p $@

$(BUILD)/$(BOOK).pdf: $(META) $(CHAPTERS) $(BIB) | $(BUILD)
    $(PANDOC) --citeproc \
        --bibliography=$(BIB) \
        --pdf-engine=typst \
        --toc --toc-depth=2 \
        --number-sections \
        $(META) $(CHAPTERS) \
        -o $@
    @printf '[PDF] %s\n' $@

$(BUILD)/$(BOOK).html: $(META) $(CHAPTERS) $(BIB) | $(BUILD)
    $(PANDOC) --citeproc \
        --bibliography=$(BIB) \
        --standalone \
        --toc --toc-depth=2 \
        --number-sections \
        --css=styles/web.css \
        $(META) $(CHAPTERS) \
        -o $@
    @printf '[HTML] %s\n' $@

$(BUILD)/$(BOOK).epub: $(META) $(CHAPTERS) $(BIB) | $(BUILD)
    $(PANDOC) --citeproc \
        --bibliography=$(BIB) \
        --split-level=1 \
        --toc \
        --epub-cover-image=assets/cover.jpg \
        --css=styles/epub.css \
        $(META) $(CHAPTERS) \
        -o $@
    @printf '[EPUB] %s\n' $@

pdf: $(BUILD)/$(BOOK).pdf
html: $(BUILD)/$(BOOK).html
epub: $(BUILD)/$(BOOK).epub

clean:

```

```
rm -rf $(BUILD)
```

Running `make all` produces all three formats; running `make pdf` produces only the PDF. When any chapter file or the metadata changes, only the dependent outputs are rebuilt.

Heading levels and `shift-heading-level-by`

When the Pandoc source uses `#` for chapter-level headings, the mapping is already sensible for books: `#` is the chapter level, `##` the section level, `###` the subsection level.

If the source uses `##` for the top level (as in an article structure), shift the heading levels with:

```
pandoc ... --shift-heading-level-by=-1 ...
```

This promotes all headings by one level: `##` becomes `\chapter`, `###` becomes `\section`, and so on.

The colophon

The colophon is the book's closing statement about its own production. It is a centuries-old tradition — early printers used it to identify themselves and date the work — and it remains appropriate for books produced with deliberate craft. For a CLI-produced book, the colophon is an opportunity to document the toolchain and typefaces used:

Colophon

This book was set from Markdown using Pandoc and Typst.
Body text is set in EB Garamond 11 point with 14 points of leading.
Section headings use Fira Sans. Code examples are set in JetBrains Mono.

The Makefile that produced this book contains fewer than fifty lines. It runs in under ninety seconds on a contemporary laptop.

Source files are available at <https://github.com/example/cli-typographer>.

In a Typst-backed book, the colophon is simply a final unnumbered page styled by the template:

```
#pagebreak(to: "odd")
#align(center + horizon)[
  #text(size: 9pt, style: "italic")[Colophon.]
  #v(6pt)
  This book was set from Markdown using Pandoc and Typst.
]
```

This chapter completes Part IV's gallery of document types. The book is the most complex document in the collection, and the complete project — chapters, front matter, back matter, index, bibliography, multi-format output, automated build — brings together every technique from Parts I through III.

Part V turns to refinement: the techniques for improving typographic quality beyond correct production, including microtypography, complex table layouts, multilingual documents, and the fine details that separate adequate typography from excellent typography.

Craft and Refinement

Tables and Complex Layouts

Tables are among the most typographically demanding elements in document production. They involve simultaneous decisions about column widths, alignment, spacing, rules, captions, and placement — and they must remain readable, not merely correct. In a Markdown-first workflow, tables are also where authors are most tempted to abandon the source-neutral path too early. The right response is not to jump straight into raw LaTeX, but to decide carefully which layer should own the complexity: Markdown or CSV for simple data, CSS or Quarto for web presentation, and Typst or another PDF backend only when page-specific control is truly needed.

This chapter therefore treats Markdown, HTML/CSS, and Typst-style backend logic as the default path, with LaTeX packages covered as backend-specific techniques when you are already committed to a LaTeX template. The same principle applies to spanning layouts such as sidebars, marginalia, callout boxes, and wrapfigures.

Markdown and backend-aware table workflows

Start with the simplest representation that preserves the data cleanly. Plain Markdown tables and CSV-backed generation are easiest to maintain in Git; when the PDF needs spanning cells, repeated headers, or strict width control, move the table logic into the print backend instead of abandoning the whole workflow. Typst's table primitives are often a better next step than backend-specific macro work.

Simple tables and grouped headers

For most documents, the source should begin as a plain Markdown table:

```
Tool	Primary output	Math	Compilation speed
Typst	PDF	Good	Fast
Pandoc	Many formats	Moderate	Fast
Quarto	Many formats	Moderate	Moderate

: CLI typesetting tools {#tbl-tools}
```

The important design rules are the same regardless of backend:

- no vertical rules
- strong distinction between header and body
- right alignment for numeric data
- consistent spacing rather than decorative lines

When grouped headers are needed, move the layout into Typst:

```
#table(
  columns: (auto, auto, auto, auto),
  align: (left, left, right, right),
  table.header(
    table.cell(colspan: 2)[*Source*],
    table.cell(colspan: 2)[*Output*],
  ),
  [Format], [Engine], [PDF], [HTML],
  [Markdown], [Pandoc + Typst], [Excellent], [n/a],
  [Markdown], [Pandoc + HTML], [Good], [Excellent],
  [Typst], [Native], [Excellent], [n/a],
)
```

Cells spanning multiple rows

Spanning rows is a layout feature, so it belongs in the PDF backend rather than in ad hoc source hacks:

```
#table(
  columns: (auto, auto, auto, auto),
  table.header([Source], [Engine], [PDF], [HTML]),
  table.cell(rowspan: 3)[Markdown],
  [Pandoc + HTML], [Good], [Excellent],
  [Pandoc + Typst], [Excellent], [n/a],
  [Pandoc + DOCX], [n/a], [n/a],
  [Typst], [Native], [Excellent], [n/a],
)
```

The table data still originates cleanly in Markdown, CSV, or YAML; the spanning behaviour is introduced only where the final page geometry is known.

Tables that fill the text width

When a table needs one descriptive column and one short label column, Typst's fractional widths are usually enough:

```
#table(
  columns: (1fr, 3fr),
  table.header([*Package*], [*Description*]),
  [table], [Structured tables with explicit column sizing and alignment.],
  [grid], [Useful when the content is logically tabular but visually
  ↪ irregular.],
  [figure], [A better wrapper when the table needs a caption and
  ↪ cross-reference.],
)
```

For HTML output, use CSS `table-layout: fixed` or `colgroup` only when the browser's default sizing produces bad results. Most tables improve more from better content and shorter prose than from heavier width control.

Multi-page tables

When a table may span multiple pages, the best strategy is usually to keep the data outside the prose source and generate the table from CSV or

JSON. In practice that means a data file such as `package-order.csv` plus a build step in Quarto or Pandoc that reads the data and renders it as a table for the target format.

A `.qmd` chapter might include an executable cell that reads the CSV and labels the resulting table; a plain Pandoc workflow can do the same with a preprocessing step that emits Markdown. The important point is that the data stays tabular and editable outside the prose manuscript.

For PDF, the template or execution layer is responsible for repeating headers and splitting pages cleanly. That keeps the source data maintainable and avoids hand-editing large tables in prose files.

Table alignment in CSS

In HTML output from Pandoc, table alignment follows the Markdown syntax: `:-` for left, `:--` for centre, `---` for right. The CSS for these tables should reinforce the booktabs aesthetic:

```
table {
  border-collapse: collapse;
  width: 100%;
  font-size: 0.95em;
}

thead tr {
  border-top: 2px solid currentColor;
  border-bottom: 1px solid currentColor;
}

tbody tr:last-child {
  border-bottom: 2px solid currentColor;
}

th, td {
  padding: 0.4em 0.75em;
  text-align: left;
}

th { font-weight: 600; }
```

```
/* Numeric columns */
td:has(+ td), th:has(+ th) { }
```

For the `text-align: right` convention on numeric columns, target specific column indices in CSS or use Pandoc’s attribute mechanism to add classes to specific columns.

Sidebars and callout boxes

Sidebars — blocks of supplementary text set apart from the main column — appear in technical books, textbooks, and reference manuals. In a Markdown-first workflow, the source should normally express them as fenced divs or Quarto callouts and leave the presentation to the back-end:

```
::: {.note}
Choose a PDF engine and font stack that actually support the scripts in
↔ your document.
:::
```

In HTML, CSS can render that as a sidebar or callout. In Typst, the same semantic class can map to a boxed block with a pale fill and stronger border. The key decision is that the source says “this is a note”, not “draw a blue rectangle here”.

Marginalia

Margin notes are best treated as a layout variant of an aside. In Quarto, the source can mark them with margin classes:

```
The baseline [The invisible line on which letters rest.]{.column-margin}
is the fundamental reference for vertical alignment.
```

In HTML, CSS places that content in the margin at wide breakpoints and in the body flow at narrow ones. In PDF, a Typst template can position the aside in the outer margin when the page size permits it.

Wrapping text around figures

Wrapped figures are usually better handled in HTML with `float` or grid-based CSS than in print PDFs, where they often create awkward white space. When you do need the effect in a web publication:

```
.wrap-figure {  
  float: right;  
  width: 38%;  
  margin: 0 0 0.8rem 1rem;  
}
```

Use wrapped figures only for short illustrations embedded in running text. For anything substantial, a normal block figure is almost always cleaner.

Templates and Style Systems

The most valuable investment in any document production workflow is the template: the system that encodes typographic decisions once and applies them everywhere. A well-designed template means that producing a new document in an established style requires no design decisions — the document inherits its appearance from the template, the author focuses on content, and the result is consistent across every document the template produces.

This chapter covers template development at three levels: reusable PDF style layers, Pandoc or Quarto templates and defaults, and CSS custom property systems for web output. For a modern CLI workflow, the order of preference should usually be Markdown source first, Typst for print-ready PDF work second, and HTML/CSS or EPUB stylesheets for the web-facing formats.

Building a reusable PDF style layer

If your PDF path is Typst, the equivalent of a house style is an imported module or package that sets page, text, heading, figure, and table defaults once and exposes only a few controlled parameters. The principle matters more than the backend: authors should edit Markdown and metadata, not layout code.

Module structure

```
// house.typ

#let palette = (
  text: rgb("#1c1c1c"),
  accent: rgb("#1a4e8c"),
  border: rgb("#d7dce2"),
)

#let article-style(
  paper: "a4",
  margin: (x: 30mm, y: 25mm),
  font: "EB Garamond",
  size: 11pt,
  columns: 1,
) = {
  set page(paper: paper, margin: margin)
  set text(font: font, size: size, fill: palette.text)
  set par(justify: true)

  show heading.where(level: 1): it => block(above: 1.5em, below: 0.6em)[
    #set text(weight: "bold", size: 18pt, fill: palette.accent)
    #it
  ]

  if columns == 2 {
    set page(columns: 2)
  }
}

#let cli(text) = raw(text)
#let pkg(text) = text(font: "Source Sans 3", text)
#let file(text) = text(font: "JetBrains Mono", text)
```

Save this in `styles/house.typ` and import it from individual templates or one-off documents:

```
#import "styles/house.typ": article-style, cli, pkg, file

#article-style()
```

All typographic defaults are centralised. To change the heading style for every document using the house module, edit `article-style` once.

Adding controlled options

Per-document variations should be explicit parameters rather than ad hoc template edits:

```
#article-style(
  font: "Source Serif 4",
  columns: 2,
  margin: (x: 18mm, y: 20mm),
)
```

That is the right style discipline for any modern template system: narrow, documented, and intentionally limited.

Imports and responsibilities

A shared style module should set defaults and expose a small public API. Individual documents should import it and supply content. That division of labour matters more than the syntax of any one backend.

Pandoc templates

Pandoc templates are text files that control the structure of the output document. Quarto defaults and format blocks play a similar role. They are covered in Chapter 6, but their role in style systems deserves elaboration here.

A template for a custom Pandoc workflow should be treated as a versioned, maintained artifact — not a one-time customisation. Store it alongside the Typst modules and stylesheets:

```
styles/
□□ article.typ      ← Typst template for PDF
□□ book.typ        ← Typst template for books
□□ web.html        ← HTML template
□□ epub.css        ← EPUB stylesheet
```

Backend templates as style controllers

A Pandoc or Quarto template's job is to translate metadata into a coherent backend-specific document. In Typst, that usually means importing a shared module and binding metadata to a small public API. For a style system, the template should load the house layer and expose only the variables that should be customisable per-document:

```
#import "house.typ": article-style

#let render(meta, body) = {
  article-style(
    font: meta.mainfont.or("EB Garamond"),
    size: meta.fontsize.or(11pt),
  )

  if meta.title != none [
    align(center)[
      #text(size: 22pt, weight: "bold")[#meta.title]
      #if meta.subtitle != none [
        #v(4pt)
        #text(size: 12pt, style: "italic")[#meta.subtitle]
      ]
      #v(6pt)
      #meta.author
    ]
    #v(12pt)
  ]

  body
}
```

This template delegates design decisions to `house.typ` while exposing only a few deliberate overrides. Authors who want the standard look simply write in Markdown; authors with specific requirements can override individual settings in metadata.

The HTML template as a site framework

For a documentation site or web publication with a consistent visual identity, a Pandoc HTML template controls the page structure:

```

<!DOCTYPE html>
<html lang="$lang$" >
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>$if(title-prefix)$title-prefix$ - $endif$$pagetitle$</title>
  <link rel="stylesheet"
    ↪ href="$if(css-root)$$css-root$$else$/$endif$$styles/main.css">
    $for(header-includes)$$header-includes$$endif$
</head>
<body>
  <header class="site-header">
    <a class="site-title" href="/">$if(site-title)$$site-title$$endif$</a>
    <nav>$if(site-nav)$$site-nav$$endif$</nav>
  </header>
  <main>
    $if(title)$
    <header class="page-header">
      <h1>$title$</h1>
      $if(subtitle)$<p class="subtitle">$subtitle$</p>$endif$
      $if(author)$<p class="author">$for(author)$$author$$sep$,
    ↪ $endif$</p>$endif$
    </header>
    $endif$
    $if(toc)$
    <nav class="toc">$table-of-contents$</nav>
    $endif$
    <article>$body$</article>
  </main>
  <footer>$if(site-footer)$$site-footer$$endif$</footer>
</body>
</html>

```

Site-level variables (site-title, site-nav, site-footer) are defined in the Pandoc defaults file and apply to every page; page-level variables (title, author) are defined in individual documents.

CSS custom property systems for web typography

A CSS custom property (CSS variable) system translates design decisions into maintainable, consistent code. Rather than scattering font-

family: 'EB Garamond', Georgia, serif across dozens of CSS rules, define it once as a variable and reference it everywhere. When the typeface changes, one line changes.

The design token layer

```
/* _tokens.css – the source of truth for all design decisions */

:root {
  /* Typography */
  --font-body: 'EB Garamond', Georgia, serif;
  --font-heading: 'Fira Sans', system-ui, sans-serif;
  --font-mono: 'JetBrains Mono', 'Courier New', monospace;

  /* Type scale (Major Third, 1.25 ratio) */
  --size-xs: 0.64rem; /* ~10px */
  --size-sm: 0.8rem; /* ~13px */
  --size-base: 1rem; /* ~16px */
  --size-md: 1.25rem; /* ~20px */
  --size-lg: 1.563rem; /* ~25px */
  --size-xl: 1.953rem; /* ~31px */
  --size-2xl: 2.441rem; /* ~39px */
  --size-3xl: 3.052rem; /* ~49px */

  /* Spacing scale */
  --space-xs: 0.25rem;
  --space-sm: 0.5rem;
  --space-md: 1rem;
  --space-lg: 2rem;
  --space-xl: 4rem;

  /* Measure (line length) */
  --measure: 66ch;
  --measure-wide: 80ch;

  /* Line heights */
  --leading-tight: 1.2;
  --leading-base: 1.6;
  --leading-loose: 1.8;

  /* Colours */
  --color-text: #1c1c1c;
```

```
--color-muted:    #555;
--color-accent:   #1a4e8c;
--color-bg:       #ffffff;
--color-bg-alt:   #f8f8f8;
--color-border:   #ddd;
}
```

The component layer

```
/* _base.css - base styles using tokens */

html { font-size: 18px; }

body {
  font-family: var(--font-body);
  font-size: var(--size-base);
  line-height: var(--leading-base);
  color: var(--color-text);
  background: var(--color-bg);
  max-width: var(--measure);
  margin: 0 auto;
  padding: var(--space-md) var(--space-lg);
}

h1, h2, h3, h4, h5, h6 {
  font-family: var(--font-heading);
  font-weight: 500;
  line-height: var(--leading-tight);
  color: var(--color-text);
}

h1 { font-size: var(--size-3xl); }
h2 { font-size: var(--size-2xl); }
h3 { font-size: var(--size-xl); }
h4 { font-size: var(--size-lg); font-style: italic; font-weight: 400; }

code, pre {
  font-family: var(--font-mono);
  font-size: var(--size-sm);
}
```

Dark mode with CSS custom properties

A minimal dark mode that swaps the design tokens without rewriting the component rules:

```
@media (prefers-color-scheme: dark) {
  :root {
    --color-text: #e8e8e8;
    --color-muted: #aaa;
    --color-accent: #7ba7d4;
    --color-bg: #1a1a1a;
    --color-bg-alt: #242424;
    --color-border: #444;
  }
}
```

All component rules remain unchanged — they reference variables, not colours directly. The dark mode override replaces only the token definitions.

House style consistency

A style system is only as good as its enforcement. For a team or organisation producing multiple documents, enforcing house style requires:

A shared template repository: store `.typ` files, Pandoc templates, and CSS in a shared Git repository. Documents import from this repository rather than copying files locally.

A defaults file per document type: the Pandoc defaults file specifies the template, the CSS, and the allowed metadata variables:

```
# defaults/house-article.yaml
from: markdown
to: html5
template: templates/article.html
css: https://styles.example.com/main.css
metadata:
  lang: en-GB
```

Linting: a Pandoc Lua filter can enforce metadata requirements — checking that title, author, and date are always present, that lang is always set, and that unknown metadata keys are flagged. A simple filter:

```
-- check-metadata.lua
function Meta(meta)
  local required = {"title", "author", "date"}
  for _, key in ipairs(required) do
    if not meta[key] then
      io.stderr:write("Warning: missing required metadata: " .. key ..
        ↵ "\n")
    end
  end
  return meta
end
```

```
pandoc document.md --lua-filter=check-metadata.lua -o output.html
```

This filter runs on every build and warns about missing metadata without failing the build — configurable to `os.exit(1)` if hard enforcement is needed.

Multilingual and Non-Latin Typesetting

Most of this book has assumed documents written in a single language using the Latin alphabet. That assumption covers a wide range of practical work, but not all of it. Technical documentation is routinely produced in German, French, Arabic, Chinese, Japanese, and dozens of other languages. Academic papers cite sources in non-Latin scripts. Books are translated. The CLI typographer working in an international context needs to know how to handle scripts beyond Latin, bidirectional text, and documents that mix multiple languages.

This chapter covers the three main contexts: multilingual Latin-script documents (multiple Western European languages), bidirectional text (Arabic and Hebrew), and CJK typesetting (Chinese, Japanese, Korean). The source-level rule is simple: keep language and direction metadata in the Markdown, then choose a PDF backend with the right fonts and shaping support. Typst and modern browser engines alike depend on correct Open Type fonts; neither can rescue a source file that fails to mark language boundaries clearly.

Multilingual Latin-script documents

When a document mixes English with French, German, Spanish, or other Latin-script languages, the primary concern is correct language metadata and hyphenation across the whole pipeline. In a Markdown-first workflow, set the document language in front matter and tag inline spans or divs when the language changes.

```
---  
lang: en-GB  
---
```

For block-level switches:

English body text with the document's default language.

```
::: {lang=fr}
Ce texte est en français. La césure et les règles de ponctuation
doivent suivre les conventions françaises.
:::
```

For inline switches:

The French term `[mise en page]{lang=fr}` refers to the layout of a printed page.

The PDF backend then needs fonts and shaping support that match the declared languages. In Typst, that usually means setting a fallback font family list rather than switching the source language:

```
#set text(
  font: ("EB Garamond", "Source Serif 4", "Noto Serif"),
  lang: "en",
)
```

Smart quotation marks

Different languages use different quotation marks: English uses “double curly quotes” and ‘single curly quotes’; German uses „Gänsefüßchen“; French uses « guillemets ».

Pandoc's smart typography or Quarto's default reader extensions handle the document's primary language well. For inline language switches, mark the language in the source and verify the output manually:

English quotation: “This is an English quotation.”

German quotation: [„Das ist ein deutsches Zitat.“]{lang=de}

Bidirectional text: Arabic and Hebrew

Arabic and Hebrew are written right-to-left. When mixed with left-to-right text (Latin script, numbers), the result is *bidirectional text* — a layout challenge that requires both the font shaping engine and the layout engine to understand Unicode’s bidirectional algorithm. In a Markdown-first workflow, the first responsibility is still source metadata: get `lang` and `dir` right before you worry about backend-specific commands.

In a Markdown-first workflow, the first responsibility is still metadata:

```
---
lang: ar
dir: rtl
---
```

For mixed-direction documents, mark the local direction and language where they change:

This is English text. The following is Arabic:

```
::: {lang=ar dir=rtl}
٠٠٠ ٠٠٠٠ ٠٠٠٠٠ ٠٠٠٠٠٠٠ ٠٠٠٠٠٠٠٠ ٠٠ ٠٠٠٠٠٠ ٠٠٠ ٠٠٠٠٠٠٠.
:::
```

Mixing [٠٠٠٠٠٠٠]{lang=ar dir=rtl} inline with English.

In Typst, choose an Arabic-capable font and set the language on the relevant span or block:

```
#set text(font: ("Amiri", "Noto Naskh Arabic", "EB Garamond"))
```

Arabic and Hebrew typesetting additionally require:

Text shaping: Arabic letters are context-sensitive — the same letter has different forms at the start, middle, and end of a word, and when isolated. Correct shaping is handled by modern shaping engines such as

HarfBuzz. Any OpenType Arabic font with a complete glyph set will shape correctly.

Appropriate fonts: Free options include Amiri (classical calligraphic Arabic), Scheherazade (for diacritical texts), and Noto Sans Arabic. These are designed to the standards of Arabic typography and include all required ligatures.

Numbers: Arabic text typically uses Arabic-Indic numerals (٠١٢٣٤٥٦٧٨٩) or Western-Arabic numerals depending on context and register. The chosen layout engine and font stack should respect the language and script metadata when selecting numeral forms.

For Pandoc documents in Arabic, set the language and direction in the metadata:

```
---  
lang: ar  
dir: rtl  
---
```

Pandoc generates the appropriate HTML `dir="rtl"` attribute and passes the language information through to the selected output backend.

CJK typesetting: Chinese, Japanese, Korean

CJK (Chinese, Japanese, Korean) typesetting involves a set of challenges distinct from both Latin and bidirectional typesetting: the character sets are large (tens of thousands of glyphs), the spacing rules differ from Western practice, and the line-breaking rules prohibit certain characters from appearing at the start or end of a line.

For documents with occasional CJK characters embedded in primarily Latin text, the main requirement is a font stack with appropriate fallback:

```
---  
lang: zh-Hans  
mainfont: "EB Garamond"  
cjkfont: "Noto Serif CJK SC"  
---
```

In Typst, declare a fallback list that includes the relevant CJK family:

```
#set text(font: ("EB Garamond", "Noto Serif CJK SC"))
```

For Japanese or Korean documents, switch the primary language metadata and choose the corresponding font family:

```
---  
lang: ja  
mainfont: "Source Serif 4"  
cjkfont: "Noto Serif CJK JP"  
---
```

For HTML output, CJK text requires no special handling beyond the `lang` attribute (which Pandoc sets from the metadata): web browsers select appropriate system fonts for CJK characters automatically.

OpenType language features

Beyond script support, OpenType fonts provide language-specific typographic features activated by the BCP 47 language tag. For example, some Cyrillic characters have different glyph forms for Russian versus Bulgarian; some Latin characters have language-specific alternates for Dutch, Romanian, and Catalan.

In Typst and modern browser engines, language tagging is the primary trigger:

```
#set text(font: ("Gentium Plus", "Noto Serif"), lang: "nl")
```

That affects glyph selection only when the font actually ships language-specific alternates. Hyphenation remains a separate responsibility controlled by the language metadata and the layout engine.

In CSS, language-specific OpenType features use `font-language-override`:

```
:lang(nl) {  
  font-language-override: "NLD"; /* Dutch */  
}  
  
:lang(bg) {  
  font-language-override: "BGR"; /* Bulgarian Cyrillic */  
}
```

The four-character OpenType language system tags differ from BCP 47 language codes. The full mapping is in the OpenType specification; the important practical point is to keep the source language tagging correct and then verify that the chosen font actually contains the expected language-specific forms.

Fine Typography

The preceding chapters have covered what must be done to produce a correct document. This chapter covers what can be done to produce an excellent one. The gap between correct and excellent is the territory of fine typography: microtypography, optical margin alignment, the elimination of widows and orphans, the treatment of special punctuation, and the attention to the paragraph as the unit of visual design.

None of this is visible at a glance. A reader does not notice that a paragraph's lines are justified to within a hair's breadth of perfection, or that the hanging punctuation extends slightly into the margin to maintain the visual alignment of the text block. What they notice is an absence — an absence of friction, of awkward line breaks, of the tiny disturbances that accumulate, in poorly typeset text, into a generalised sense of difficulty. Fine typography removes those disturbances.

In a Markdown-first workflow, fine typography is layered. Some decisions belong in the source itself: smart punctuation, non-breaking spaces, disciplined heading structure, and restraint about manual overrides. Some belong in CSS or the HTML template. Some belong in the PDF backend, where Typst can refine justification, line breaking, and spacing for print. The point of this chapter is to show where each layer earns its keep.

Backend-specific microtypography

The available controls differ by engine. Typst, print-oriented CSS, and browser layout engines do not expose the same knobs, and that is acceptable: the workflow should stay source-centred even when the backends

diverge. In a modern workflow, the main microtypographic decisions are:

- choosing a good text face with real italics, small caps, and ligatures
- enabling justified text only when the measure is suitable
- tagging the document language correctly for hyphenation
- applying tracking only to display text, small caps, and running heads

Edge alignment and hanging punctuation

The visual edge of a paragraph should look straight even when lines end with commas or quotation marks. On the web, the only standard control is CSS hanging punctuation:

```
p,
blockquote {
  hanging-punctuation: first last;
}
```

Support is still uneven, so this should be treated as an enhancement rather than a guarantee.

In Typst, edge handling is mostly implicit. The more important practical decision is to avoid bad measures and badly chosen fonts, which create visible edge noise that no low-level setting can rescue.

Justification quality

Typst does not ask the author to micromanage glyph expansion. The right equivalent is to set up the paragraph environment correctly and then inspect the pages:

```
#set page(margin: (x: 28mm, y: 24mm))
#set text(font: "New Computer Modern", size: 11pt, lang: "en")
#set par(justify: true)
```

If justification looks uneven, solve the problem at the layout level first: shorten the measure, revise the copy, or choose a more suitable typeface. That is usually more effective than hunting for backend-specific tuning parameters.

Tracking

Tracking is the uniform adjustment of letter-spacing for specific text elements. It should be applied narrowly, not globally:

```
#show smallcaps: set text(tracking: 0.03em)

#set heading(numbering: "1.")
#show heading.where(level: 1): set text(tracking: -0.01em)
```

Tracking small capitals slightly more openly is a classical typographic convention. Large headings often benefit from a very slight negative track.

A practical Typst baseline

A reasonable print baseline for prose:

```
#set text(font: "Source Serif 4", size: 11pt, lang: "en")
#set par(justify: true, leading: 0.68em)
#set heading(numbering: "1.")
#show smallcaps: set text(tracking: 0.025em)
```

That is not a magic formula. It is simply a sensible starting point that assumes a good text face, a comfortable measure, and language metadata that allows the engine to hyphenate correctly.

Optical margin alignment

Optical margin alignment is the practice of hanging punctuation and certain character edges slightly outside the text block so that the visual margin appears straight. In English prose, the most important characters to hang are:

- Quotation marks (both opening and closing)
- Hyphens in compound words that break at line endings
- Periods, commas, and other punctuation at line ends

In dedicated PDF engines, this is mostly handled by the compositor rather than by source markup. For display-size text — large headings where individual characters have significant visual weight — manual fine-tuning may still be needed.

In CSS, `hanging-punctuation: first last` enables hanging punctuation for browsers that support it. Support is currently limited to Safari, but the property degrades gracefully:

```
p {  
  hanging-punctuation: first last;  
}
```

The `first` value hangs punctuation at the start of the first line of a block; `last` hangs it at the end of the last line. For pull quotes and display text where the alignment is prominent, this is worth enabling.

Widows and orphans

A *widow* is the last line of a paragraph stranded alone at the top of a page. An *orphan* is the first line of a paragraph stranded alone at the bottom. Both are typographic failures: they interrupt the reading flow and leave an awkward amount of white space either before or after the isolated line.

In CSS, the available controls are:

```
p {  
  widows: 2; /* minimum lines at top of page after break */  
  orphans: 2; /* minimum lines at bottom of page before break */  
}
```

CSS widows and orphans are respected in print stylesheets and PDF generation via WeasyPrint, though not by all PDF engines.

For Typst and other PDF engines, the practical workflow is editorial rather than parametric: inspect the pages, adjust the text, or insert a small amount of vertical flexibility in the layout. Widows and orphans are usually symptoms of a page-design problem, not just a penalty-setting problem.

Hyphenation control

Automatic hyphenation in dedicated layout engines is usually excellent but not infallible. Technical terms, proper nouns, and words in minority languages may be hyphenated incorrectly or not at all.

Suppressing bad breaks for a word or phrase:

```
Dr&nbsp;Smith  
11&nbsp;pt
```

For HTML, a non-breaking space is often the right fix. For PDF-oriented workflows, the better fix is usually to rewrite the line rather than force the engine into narrower rules.

Suggesting a break point for difficult words:

```
typo&shy;graphy  
micro&shy;type
```

The soft hyphen becomes visible only if the line actually breaks there.

The paragraph as the unit of design

Paragraph-level line breaking is one of the clearest advantages of dedicated layout engines over ordinary word processors. Good layout engines consider the paragraph as a whole rather than choosing each line in isolation, which is one reason high-quality PDF workflows justify better than office software.

The point is not to tune hidden engine thresholds unless you are debugging a specific backend. The point is to think paragraph by paragraph: if the measure is too narrow, the font too large, or the wording too rigid, the paragraph will look bad regardless of the engine.

In practice, the best fixes are usually:

- shorten or rewrite the sentence
- widen the measure slightly
- reduce heading clutter above the paragraph
- move an image or note that is constraining the text block

Letterspacing and small capitals

Letterspacing (tracking) is appropriate in specific contexts:

- **Small capitals:** as noted above, a small positive track of 25–50 units improves the texture of all-small-capital text.
- **Headings in all-capitals:** capitals set at display size benefit from positive tracking. A typical value is 100–150 units.
- **Running headers:** tight small capitals or tracked uppercase text is conventional for running headers in serious typographic work.

Do not letterspace body text. The claim that letterspacing improves legibility for lowercase body text is not supported by reading research, and letterspacing lowercase breaks the visual cohesion of words. Track display text; leave body text at its designed tracking.

In CSS, `letter-spacing` applies tracking in em units:

```
.small-caps {
  font-variant: small-caps;
  letter-spacing: 0.04em;
}

h1, h2 {
  font-family: var(--font-heading);
  letter-spacing: -0.01em; /* slight tight track for large headings */
}
```

Note that large headings typically benefit from *negative* tracking — slightly tighter than the font’s default — because at display sizes the normal spacing appears too loose.

Typographic details in Markdown and Pandoc output

Several typographic refinements apply to Pandoc’s output specifically.

Smart quotes: Pandoc’s `+smart` extension converts straight apostrophes and quotation marks to typographic equivalents. Enable it with:

```
pandoc -f markdown+smart ...
```

Or set it in a defaults file:

```
from: markdown+smart
```

Non-breaking spaces: In running text, certain character combinations should not be broken across lines: a number and its unit (“11 pt”), a title and a name (“Dr Smith”), the last two words of a paragraph. Pandoc passes through HTML non-breaking spaces () from Markdown source. For HTML output, CSS `white-space: nowrap` can be applied to specific spans.

Dashes: Pandoc’s smart extension converts double hyphens (--) to en dashes (–) and triple hyphens (---) to em dashes (—). This is the correct

handling: never use double hyphens where an en dash or em dash is intended, and never use the minus sign key where a dash is intended.

Ellipsis: Three periods (. . .) in smart mode become the ellipsis character (...), which is slightly different in spacing from three periods in sequence. In a high-quality typeset document, use the dedicated character.

Fine typography is the difference between a document that communicates and a document that communicates with authority. The techniques in this chapter do not change what a document says; they change how much confidence the reader places in the saying. A well-typeset document feels authoritative. A carelessly typeset document, no matter how good its content, carries a faint suggestion of carelessness throughout.

This is the work: to make the form worthy of the content. The tools are now in your hands.

Appendix A: Quick Reference — Pandoc Flags

This appendix is a concise reference to the Pandoc flags most commonly needed in document production workflows. Flags are grouped by purpose. For the complete documentation, run `pandoc --help` or consult pandoc.org/MANUAL.html.

Input and output

| Flag | Long form | Effect |
|------------------------|----------------------------|--|
| <code>-f FORMAT</code> | <code>--from=FORMAT</code> | Specify input format |
| <code>-t FORMAT</code> | <code>--to=FORMAT</code> | Specify output format |
| <code>-o FILE</code> | <code>--output=FILE</code> | Write output to file
(default: <code>stdout</code>) |
| | <code>--file-scope</code> | Parse each file
independently (useful for
multi-file projects) |
| | <code>--sandbox</code> | Disable features that read
from the filesystem |

Common input formats: `markdown`, `gfm`, `commonmark`, `html`, `latex`, `docx`, `epub`, `rst`, `org`, `typst`

Common output formats: `html`, `html5`, `pdf`, `latex`, `beamer`, `epub`, `epub3`, `docx`, `revealjs`, `man`, `plain`, `markdown`, `chunkedhtml`

Document structure

| Flag | Long form | Effect |
|------|----------------------------|--|
| -s | --standalone | Produce complete document (with header/footer) |
| | --toc | Add table of contents |
| | --toc-depth=N | TOC depth (default: 3) |
| -N | --number-sections | Number headings |
| | --number-offset=N[,N...] | Starting number for section numbering |
| | --top-level-division=TYPE | Map top heading to section, chapter, or part |
| | --shift-heading-level-by=N | Shift all heading levels by N (positive or negative) |
| | --split-level=N | Heading level that splits chunkedhtml/EPUB files |

Metadata and variables

| Flag | Long form | Effect |
|--------------|----------------------|--|
| -M KEY:VALUE | --metadata=KEY:VALUE | Set or override a metadata variable |
| | --metadata-file=FILE | Read metadata from YAML file |
| -V KEY:VALUE | --variable=KEY:VALUE | Set a template variable (not metadata) |
| -d FILE | --defaults=FILE | Load defaults from YAML file |

Difference between -M and -V: -M sets metadata accessible with `$meta-json$` and used by `--ci tproc`; -V sets template variables only, bypassing the metadata system.

Templates and styling

| Flag | Long form | Effect |
|------------------------|--|---|
| | <code>--template=FILE</code> | Use custom template |
| <code>-c URL</code> | <code>--css=URL</code> | Link CSS stylesheet (HTML/EPUB) |
| <code>-H FILE</code> | <code>--include-in-header=FILE</code> | Insert file content in document head |
| <code>-B FILE</code> | <code>--include-before-body=FILE</code> | Insert before document body |
| <code>-A FILE</code> | <code>--include-after-body=FILE</code> | Insert after document body |
| <code>-D FORMAT</code> | <code>--print-default-template=FORMAT</code> | Print the default template for a format |
| | <code>--reference-doc=FILE</code> | Use reference DOCX or ODT for styles |

PDF generation

| Flag | Long form | Effect |
|------|--------------------------------------|--|
| | <code>--pdf-engine=ENGINE</code> | PDF engine: pdflatex, xelatex, lualatex, wkhtmltopdf, weasyprint |
| | <code>--pdf-engine-opt=STRING</code> | Pass option to the PDF engine |

Useful PDF metadata variables (via `-M` or `YAML`):

| Variable | Effect |
|----------------------------|---|
| <code>documentclass</code> | LaTeX class (article, book, report, scrartcl) |
| <code>classoption</code> | Class options list |
| <code>geometry</code> | Page geometry string passed to geometry package |

| Variable | Effect |
|-------------|---------------------------------------|
| fontsize | Base font size (10pt, 11pt, 12pt) |
| mainfont | Main typeface (XeLaTeX/LuaLaTeX only) |
| sansfont | Sans-serif typeface |
| monofont | Monospace typeface |
| linestretch | Line spacing multiplier (1.25, 1.5) |
| colorlinks | Colour hyperlinks (true/false) |
| linkcolor | Internal link colour (e.g. NavyBlue) |
| urlcolor | URL link colour |
| hidelinks | Remove link formatting entirely |
| CJKmainfont | CJK main font (XeLaTeX) |

Citations and bibliography

| Flag | Long form | Effect |
|------|-------------------------------|---------------------------------------|
| -C | --citeproc | Process citations |
| | --bibliography=FILE | Bibliography file (.bib, .json, etc.) |
| | --csl=FILE | Citation Style Language file |
| | --natbib | Use natbib for LaTeX output |
| | --biblatex | Use biblatex for LaTeX output |
| | --citation-abbreviations=FILE | Journal abbreviation list |

Filters and extensions

| Flag | Long form | Effect |
|------------|----------------------------|---|
| -F PROGRAM | --filter=PROGRAM | External filter (JSON, older mechanism) |
| -L FILE | --lua-filter=FILE | Lua filter (recommended) |
| | --list-extensions[=FORMAT] | List available extensions for a format |

Adding/removing Markdown extensions:

```
pandoc -f markdown+smart-pipe_tables # add smart, remove pipe_tables
pandoc -f markdown+yaml_metadata_block # ensure YAML metadata is on
```

Code highlighting

| Flag | Long form | Effect |
|------|--|--------------------------------|
| | <code>--highlight-style=STYLE</code> | Code highlight style |
| | <code>--no-highlight</code> | Disable syntax highlighting |
| | <code>--syntax-definition=FILE</code> | Add custom language definition |
| | <code>--list-highlight-styles</code> | List available styles |
| | <code>--list-highlight-languages</code> | List supported languages |
| | <code>--print-highlight-style=STYLE</code> | Print style as JSON |

Built-in styles: `pygments` (default), `tango`, `espresso`, `zenburn`, `kate`, `monochrome`, `breezedark`, `haddock`

EPUB options

| Flag | Long form | Effect |
|------|--------------------------------------|-------------------------------------|
| | <code>--epub-cover-image=FILE</code> | Cover image |
| | <code>--epub-metadata=FILE</code> | Additional Dublin Core metadata XML |
| | <code>--epub-embed-font=FILE</code> | Embed font file in EPUB |
| | <code>--epub-title-page=BOOL</code> | Generate title page (true/false) |
| | <code>--epub-subdirectory=DIR</code> | Subdirectory name for EPUB content |

Diagnostics and utilities

| Flag | Long form | Effect |
|-----------------|------------------------------------|-----------------------------------|
| | <code>--verbose</code> | Print diagnostic output to stderr |
| | <code>--quiet</code> | Suppress warnings |
| | <code>--fail-if-warnings</code> | Exit with error on any warning |
| | <code>--log=FILE</code> | Write log to file (JSON format) |
| | <code>--trace</code> | Print trace messages |
| | <code>--list-input-formats</code> | List all supported input formats |
| | <code>--list-output-formats</code> | List all supported output formats |
| <code>-v</code> | <code>--version</code> | Print version information |
| <code>-h</code> | <code>--help</code> | Print help |

Commonly used combinations

Standalone HTML with TOC and numbering:

```
pandoc input.md --standalone --toc --number-sections \
  --css=style.css -o output.html
```

PDF via Typst:

```
pandoc input.md --pdf-engine=typst \
  -M mainfont="EB Garamond" \
  -M geometry="margin=25mm" \
  -o output.pdf
```

PDF via XeLaTeX with custom fonts when LaTeX compatibility is required:

```
pandoc input.md --pdf-engine=xelatex \
  -M mainfont="EB Garamond" \
  -M geometry="margin=25mm" \
  -o output.pdf
```

EPUB with cover and embedded font:

```
pandoc input.md --toc --split-level=1 \  
  --epub-cover-image=cover.jpg \  
  --epub-embed-font=fonts/EBGaramond-Regular.ttf \  
  --css=styles/epub.css \  
  -o output.epub
```

Multi-file book build:

```
pandoc metadata.yaml chapters/*.md \  
  --citeproc --bibliography=refs.bib \  
  --pdf-engine=typst \  
  --toc --number-sections \  
  -o book.pdf
```

Beamer slides with theme:

```
pandoc slides.md -t beamer \  
  --pdf-engine=xelatex \  
  -M theme=metropolis \  
  -M aspectratio=169 \  
  -o slides.pdf
```


Appendix B: Essential LaTeX Packages

This appendix lists the packages used throughout this book, organised by purpose. For each package, the table shows the recommended loading position (early = before most packages; late = after most packages), a brief description, and the chapter where it is covered in depth.

All packages listed here are included in a standard TeX Live or MiKTeX installation. Install any missing package with `tlmgr install packagename`.

Page layout and geometry

| Package | Load | Description |
|-----------------------|----------------|---|
| <code>geometry</code> | Early | Page dimensions, margins, paper size. Replaces manual <code>\hoffset</code> / <code>\voffset</code> settings. See Chapter 8. |
| <code>fancyhdr</code> | After geometry | Running headers and footers. Provides <code>\pagestyle{fancy}</code> with <code>\fancyhead</code> , <code>\fancyfoot</code> , <code>\fancyhf</code> . See Chapters 8, 20, 22. |
| <code>titlesec</code> | Any | Section heading formatting. Replaces <code>\@startsection</code> . Provides <code>\titleformat</code> and <code>\titlespacing</code> . See Chapter 19. |
| <code>setspace</code> | Any | Line spacing via <code>\setstretch{factor}</code> , <code>\onehalfspacing</code> , <code>\doublespacing</code> . Prefer <code>\linespread</code> for small adjustments. |

| Package | Load | Description |
|----------|------|---|
| parskip | Any | Replaces paragraph indentation with vertical space. Suitable for documents with short paragraphs. |
| multicol | Any | Multiple columns with automatic balancing. More flexible than the twocolumn class option. |
| wrapfig | Any | Wrap text around figures. Use with caution near page boundaries. See Chapter 23. |

Typography and fonts

| Package | Load | Description |
|-----------|---------------|---|
| fontenc | Early | Output font encoding. Always use [T1] for Western European documents with pdfLaTeX. |
| inputenc | Early | Input encoding. Use [utf8] with pdfLaTeX. Not needed with XeLaTeX or LuaLaTeX. |
| fontspec | Early | Font selection by name for XeLaTeX and LuaLaTeX. Provides <code>\setmainfont</code> , <code>\setsansfont</code> , <code>\setmonofont</code> . See Chapters 3, 12. |
| microtype | After fonts | Microtypography: character protrusion, font expansion, tracking. Major quality improvement. See Chapter 26. |
| csquotes | Any | Language-aware quotation marks. <code>\textquote{...}</code> and <code>\enquote{...}</code> . Integrates with babel/polyglossia. See Chapter 25. |
| textcomp | After fontenc | Additional text symbols: <code>\texteuro</code> , <code>\textdegree</code> , <code>\textregistered</code> . Often loaded automatically. |

Mathematics

| Package | Load | Description |
|-----------|------|---|
| amsmath | Any | Extended math environments: <code>align</code> , <code>gather</code> , <code>multline</code> , <code>cases</code> . Improved <code>\frac</code> , operators. Essential for any mathematical document. See Chapter 12. |
| amssymb | Any | Additional mathematical symbols: <code>\mathbb{R}</code> , <code>\square</code> , <code>\therefore</code> , etc. |
| amsthm | Any | Theorem environments. Provides <code>\newtheorem</code> with <code>plain</code> , <code>definition</code> , and <code>remark</code> styles. See Chapter 12. |
| mathtools | Any | Extends amsmath. Adds <code>\underbrace</code> , <code>\overbrace</code> improvements, paired delimiters. |
| siunitx | Any | Physical quantities and units: <code>\qty{300}{dpi}</code> , <code>\num{1234567}</code> . See Chapter 12. |

Tables

| Package | Load | Description |
|-----------|------|---|
| booktabs | Any | Professional table rules: <code>\toprule</code> , <code>\midrule</code> , <code>\bottomrule</code> , <code>\cmidrule</code> . See Chapter 23. |
| longtable | Any | Tables spanning multiple pages with repeating headers. Required in any Pandoc LaTeX template. See Chapter 23. |
| multirow | Any | Cells spanning multiple rows: <code>\multirow{n}{width}{text}</code> . See Chapter 23. |
| tabularx | Any | Tables that stretch to text width with auto-sizing X columns. See Chapter 23. |

| Package | Load | Description |
|------------|------|---|
| array | Any | Enhanced column specifications. Provides <code>>\cmd</code> , <code><\cmd</code> , <code>!\cmd</code> for column-level formatting. |
| tabularray | Any | Modern table package with key-value interface. Combines functionality of <code>booktabs</code> , <code>longtable</code> , <code>multirow</code> , <code>tabularx</code> . Requires recent TeX Live. |

Figures and floats

| Package | Load | Description |
|------------|---------------|---|
| graphicx | Any | Include images:
<code>\includegraphics[options]{file}</code> . Standard for all figure inclusion. See Chapter 20. |
| float | Any | Adds H placement specifier (here, definitely). Use sparingly. |
| caption | Any | Customise caption formatting.
<code>\captionsetup{font=small, labelfont=bf}</code> . |
| subcaption | After caption | Subfigures and subtables with individual captions. <code>subfigure</code> and <code>subtable</code> environments. See Chapter 20. |
| wrapfig | Any | Figures with text wrapping. See Chapter 23. |

Lists

| Package | Load | Description |
|-----------|------|--|
| enumitem | Any | Full control over list formatting. Spacing, labels, indentation. Replaces <code>enumerate</code> and <code>itemize</code> for customisation. See Chapter 12. |
| etaremune | Any | Reverse-numbered list (counts down from N). Used for publication lists in CVs. See Chapter 19. |

Cross-references and hyperlinks

| Package | Load | Description |
|----------|----------------|--|
| hyperref | Late | PDF hyperlinks, bookmarks, metadata. Load late (second-to-last). See Chapters 8, 12. |
| cleveref | After hyperref | Smart cross-references: <code>\cref{fig:x}</code> produces “Figure 1”. Load after <code>hyperref</code> . See Chapters 12, 20. |

Bibliography

| Package | Load | Description |
|----------|---------------|--|
| natbib | Any | Author-year and numeric citation styles for BibTeX. <code>\citet</code> , <code>\citep</code> , <code>\citealt</code> . See Chapters 12, 22. |
| biblatex | After fontenc | Modern bibliography system with Biber backend. More flexible than <code>natbib</code> . See Chapter 22. |

Code and verbatim

| Package | Load | Description |
|----------|------|---|
| listings | Any | Syntax-highlighted code blocks with line numbers, frames, captions. See Chapter 20. |
| minted | Any | Higher-quality syntax highlighting via Pygments. Requires <code>--shell-escape</code> . See Chapter 20. |
| verbatim | Any | Extended verbatim environment. <code>\verbatiminput{file}</code> for external files. |
| fancyvrb | Any | Customisable verbatim environments. Used by <code>minted</code> . |

Colour and boxes

| Package | Load | Description |
|-----------|------|---|
| xcolor | Any | Extended colour support: named colours, colour mixing, shade definitions. <code>\textcolor{red}{text}</code> , <code>\colorbox{yellow}{text}</code> . |
| tcolorbox | Any | Highly customisable coloured boxes, callouts, listings integration. Provides <code>\newtcolorbox</code> . See Chapter 23. |
| mdframed | Any | Framed and shaded environments that break across pages. Simpler than <code>tcolorbox</code> for basic cases. |

Language support

| Package | Load | Description |
|-------------|----------------|--|
| babel | Early | Multilingual support for pdfLaTeX: hyphenation patterns, localised strings, language environments. See Chapter 25. |
| polyglossia | Early | Multilingual support for XeLaTeX/LuaLaTeX. Better integration with fontspec. See Chapter 25. |
| xeCJK | After fontspec | Chinese, Japanese, Korean typesetting with XeLaTeX. See Chapter 25. |
| luatexja | After fonts | Japanese typesetting with LuaLaTeX. Implements JIS X 4051. See Chapter 25. |

Indexing

| Package | Load | Description |
|----------|------|---|
| makeidx | Any | Basic indexing: <code>\makeindex</code> , <code>\index{entry}</code> , <code>\printindex</code> . Requires external <code>makeindex</code> run. See Chapter 22. |
| imakeidx | Any | Enhanced indexing: multiple indexes, automatic <code>makeindex</code> invocation with <code>latexmk</code> . See Chapter 22. |

Diagrams and graphics

| Package | Load | Description |
|----------|------------|---|
| tikz | Any | Programmatic vector graphics. Extremely capable. Loads with <code>\usetikzlibrary{...}</code> for extensions. See Chapter 17. |
| pgfplots | After tikz | Data plots within LaTeX: function curves, scatter, bar charts, coordinate axes. |

Document classes (not packages)

| Class | Description |
|------------|---|
| article | Standard articles, papers, reports without chapters. |
| report | Longer documents with chapters but without book front/back matter. |
| book | Full book with parts, chapters, front matter, back matter. |
| scrartcl | KOMA-Script article. More flexible than <code>article</code> , better European defaults. |
| scrreprt | KOMA-Script report. |
| scrbook | KOMA-Script book. |
| memoir | Comprehensive class combining <code>article</code> , <code>report</code> , and <code>book</code> . Large feature set. |
| beamer | Presentation slides. See Chapter 21. |
| letter | Standard correspondence. See Chapter 18. |
| scrlettr2 | KOMA-Script letter. Superior to <code>letter</code> for professional correspondence. See Chapter 18. |
| standalone | Single-element documents (diagrams, equations). See Chapter 17. |

Recommended minimal preambles

Article (pdfLaTeX):

```
\documentclass[11pt,a4paper]{article}
\usepackage[T1]{fontenc}
\usepackage[protrusion=true,expansion=false]{microtype}
\usepackage{geometry}\geometry{margin=25mm}
\usepackage{booktabs,longtable,graphicx}
\usepackage{natbib}
\usepackage{hyperref}
\usepackage{cleveref}
```

Article (XeLaTeX):

```
\documentclass[11pt,a4paper]{article}
\usepackage{fontspec}
\setmainfont{EB Garamond}
\usepackage[protrusion=true]{microtype}
\usepackage{geometry}\geometry{margin=25mm}
\usepackage{booktabs,longtable,graphicx}
\usepackage{natbib}
\usepackage{hyperref}
\usepackage{cleveref}
```

Book (XeLaTeX, two-sided):

```
\documentclass[11pt,a4paper,twoside,openright]{book}
\usepackage{fontspec}\setmainfont{EB Garamond}
\usepackage[protrusion=true]{microtype}
\usepackage{geometry}
\geometry{
  ↪ {a4paper,inner=30mm,outer=25mm,top=30mm,bottom=25mm,
bindingoffset=10mm}
}
\usepackage{fancyhdr,booktabs,longtable,graphicx}
\usepackage{makeidx}\makeindex
\usepackage{natbib}
\usepackage[hidelinks]{hyperref}
\usepackage{cleveref}
```


Appendix C: Tool Comparison Matrix

The tables in this appendix provide a quick comparative view of the tools covered in this book. No tool is objectively best across all criteria; the correct tool depends on the specific document and workflow. Use these tables alongside Chapter 11's decision framework.

Primary tools: comprehensive comparison

The seven criteria are scored 1–5, where 5 is best. Scores represent typical usage with reasonable configuration — not theoretical limits or worst-case defaults.

| Criterion | LaTeX | Typst | Pandoc | Quarto | Org Mode | groff |
|----------------------------------|-------|-------|--------|--------|----------|-------|
| Typographic quality (PDF) | 5 | 4 | 4† | 4† | 4† | 3 |
| Multiple output formats | 1 | 1 | 5 | 5 | 4 | 2 |
| Mathematical typesetting | 5 | 4 | 3 | 4 | 3 | 2 |
| Learning curve (higher = easier) | 1 | 3 | 4 | 3 | 2 | 2 |
| Compilation speed | 2 | 5 | 4 | 3 | 3 | 5 |
| Package/extension ecosystem | 5 | 2 | 3 | 3 | 3 | 1 |
| Code execution | 1 | 1 | 1 | 5 | 4 | 1 |
| Version-control friendliness | 4 | 5 | 5 | 5 | 5 | 4 |
| Publisher acceptance | 5 | 1 | 3 | 3 | 2 | 2 |

† Via LaTeX backend

PDF output engines

| Engine | Part of | Math | System fonts | Speed | Best for |
|-------------|-------------------|--------------|-----------------|-----------|---------------------------------------|
| pdfLaTeX | TeX Live / MiKTeX | ✓ | ✗ (Type 1 only) | Fast | Compatibility, math-heavy docs |
| XeLaTeX | TeX Live / MiKTeX | ✓ | ✓ | Moderate | System fonts, multilingual |
| LuaLaTeX | TeX Live / MiKTeX | ✓ | ✓ | Slow | Lua scripting, advanced Unicode |
| Typst | Typst | ✓ | ✓ | Very fast | New projects without LaTeX dependency |
| wkhtmltopdf | Separate install | Via Math-Jax | ✓ | Moderate | CSS-driven layouts, HTML→PDF |
| WeasyPrint | Python package | Via Math-Jax | ✓ | Moderate | CSS Paged Media, print stylesheets |

Output format support by tool

A ✓ indicates native or first-class support; (✓) indicates support via an intermediate format or with configuration; ✗ indicates not supported.

| Format | LaTeX | Typst | Pandoc | Quarto | Org Mode | groff |
|---------------|-------|-------|--------|--------|----------|-------|
| PDF | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HTML | ✗ | ✗ | ✓ | ✓ | ✓ | (✓) |
| EPUB | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| DOCX | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| man page | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ |
| Beamer slides | ✓ | ✗ | ✓ | ✓ | (✓) | ✗ |
| Reveal.js | ✗ | ✗ | ✓ | ✓ | (✓) | ✗ |

| Format | LaTeX | Typst | Pandoc | Quarto | Org Mode | groff |
|--------------|-------|-------|--------|--------|----------|-------|
| ODT | ✗ | ✗ | ✓ | (✓) | ✓ | ✗ |
| Typst source | ✗ | — | ✓ | (✓) | ✗ | ✗ |
| LaTeX source | — | ✗ | ✓ | ✓ | ✓ | ✗ |

Source format compatibility

Which tools can read each source format:

| Source format | LaTeX | Typst | Pandoc | Quarto | Org Mode |
|-----------------|-------|-------|--------|--------|----------|
| Markdown (.md) | ✗ | ✗ | ✓ | ✓ | (✓) |
| Pandoc Markdown | ✗ | ✗ | ✓ | ✓ | ✗ |
| LaTeX (.tex) | ✓ | ✗ | ✓ | ✓ | (✓) |
| Typst (.typ) | ✗ | ✓ | ✓ | ✗ | ✗ |
| Org (.org) | ✗ | ✗ | ✓ | ✗ | ✓ |
| DOCX (.docx) | ✗ | ✗ | ✓ | ✓ | ✗ |
| HTML (.html) | ✗ | ✗ | ✓ | ✓ | ✗ |
| RST (.rst) | ✗ | ✗ | ✓ | ✓ | ✗ |
| QMD (.qmd) | ✗ | ✗ | ✗ | ✓ | ✗ |

Decision guide: which tool for which job

| Situation | Recommended tool | Reason |
|---------------------------|------------------|--|
| Academic paper with maths | LaTeX | Unsurpassed math quality; publisher requires it |
| Academic paper, no maths | Pandoc + Typst | Markdown source; multiple output formats with a modern PDF backend |

| Situation | Recommended tool | Reason |
|--------------------------------------|---|---|
| Journal submission
(must be .tex) | LaTeX | Publisher requirement |
| New document, no LaTeX requirement | Typst | Modern syntax; fast compilation |
| Computational report
(Python/R) | Quarto | Code execution and reproducibility |
| API documentation site | Pandoc + HTML | Multi-page HTML; version-controllable |
| Book (print + ebook) | Pandoc or Quarto + Typst | Markdown source; HTML/EPUB plus print-ready PDF |
| Presentation slides
(print) | Beamer (native or via Pandoc) | PDF; math support |
| Presentation slides
(online) | Reveal.js via Pandoc | HTML; interactive; shareable URL |
| Cover letter or correspondence | Pandoc or Typst | Markdown for multi-format; Typst for PDF-only |
| Résumé / CV
Unix man page | Markdown + Typst, or LaTeX fallback
groff -man | See Chapter 19 Standard; always available |
| Simple document, no dependencies | groff -ms | No LaTeX needed; always available |
| Emacs-based workflow | Org Mode | Native to Emacs ecosystem |

Diagram tools comparison

| Tool | Type | Output | Best for | Installation |
|----------------|----------------|--------------------------|--|---|
| TikZ | LaTeX package | PDF/SVG (via standalone) | Precise technical diagrams in LaTeX | Included in TeX Live |
| Graphviz (dot) | CLI | SVG, PDF, PNG | Automatic graph layout; dependency trees | <code>apt install graphviz</code> |
| Mermaid | JavaScript/CS | SVG, PNG | Flowcharts in Markdown; web use | <code>npm install -g @mermaid-js/mermaid-cli</code> |
| D2 | CLI | SVG | Software architecture diagrams | <code>curl -fsSL https://d2lang.com/install.sh</code>
<code>\ sh</code> |
| gnuplot | CLI | SVG, PDF, PNG, EPS | Data plots; scientific figures | <code>apt install gnuplot</code> |
| matplotlib | Python library | SVG, PDF, PNG | Data visualisation with computation | <code>pip install matplotlib</code> |
| PGFPlots | LaTeX package | PDF | Data plots within LaTeX | Included in TeX Live |

Font management tools

| Tool | Platform | Purpose |
|--------------------------|------------------------------|--|
| <code>fc-list</code> | Linux/macOS | List installed fonts; filter by properties |
| <code>fc-match</code> | Linux/macOS | Find best match for a font specification |
| <code>fc-query</code> | Linux/macOS | Inspect font file properties |
| <code>fc-cache -f</code> | Linux/macOS | Rebuild font cache after installation |
| <code>pdffonts</code> | Linux/macOS (poppler) | Verify font embedding in a PDF |
| <code>otfinfo</code> | Linux/macOS (lcdf-typetools) | Query OpenType font features |
| <code>fonttools</code> | Python (cross-platform) | Inspect, subset, and manipulate font files |

| Tool | Platform | Purpose |
|-----------|----------|----------------------------|
| Font Book | macOS | GUI font manager |
| tlmgr | TeX Live | Install TeX fonts via CTAN |

Build tool comparison

| Tool | Language | Incremental | Parallel | Learning curve | Best for |
|--------------|------------|-------------|------------|----------------|-------------------------|
| GNU Make | Shell/Make | ✓ | ✓ | Low | Any project; universal |
| latexmk | Perl | ✓ | ✗ | Very low | LaTeX-only projects |
| just | Shell | ✗ | ✓ | Low | Modern Make alternative |
| tup | C | ✓ | ✓ | Medium | Large projects |
| Shell script | Bash/sh | ✗ | Via &/wait | Low | Simple pipelines |

Appendix D: Free Font Resources

All fonts listed here are released under the SIL Open Font Licence (OFL) or an equivalent permissive licence unless otherwise noted. OFL permits use in any document — commercial or personal — and permits embedding in PDFs and EPUBs without restriction. The LPPL (LaTeX Project Public Licence) is a modified licence that permits use and distribution but restricts modification of the original.

Serif typefaces for body text

EB Garamond

Designer: Georg Duffner, updated by Octavio Pardo

Source: github.com/octaviopardo/EBGaramond

Packages: Available via Google Fonts; LaTeX package `ebgaramond`

Install: `apt install fonts-ebgaramond` or `tlmgr install ebgaramond`

A faithful revival of the types of Claude Garamont (c.1530), based on the specimen book of Conrad Berner (1592). Includes old-style figures, true small capitals, multiple ligature sets, and swash capitals. One of the finest free text faces available. Works well at 11–12pt for body text in books and articles.

Libertinus Serif

Designer: Caleb Maclennan (fork of Linux Libertine by Philipp H. Poll)

Source: github.com/alerque/libertinus

LaTeX package: `libertinus`

Install: `apt install fonts-libertinus`

A complete type family: Libertinus Serif, Sans, Mono, Display, Keyboard, and Math. The Math font makes it one of the best choices for documents with extensive mathematics that want to depart from Computer Modern. The serif face has strong, legible letterforms appropriate for long-form reading.

Source Serif

Designer: Frank Grießhammer (Adobe)

Source: github.com/adobe-fonts/source-serif

Package: Available via Google Fonts as “Source Serif 4”

A variable-font serif designed to complement Source Sans and Source Code Pro. Multiple optical sizes (Display, Text, Caption) and extensive weight range. The Text optical size is well-suited to body text at 10–12pt. Clean and functional rather than calligraphic.

Crimson Pro

Designer: Jacques Le Bailly (fork of Crimson Text by Sebastian Kosch)

Source: Google Fonts

A text-optimised humanist serif with old-style figures and small capitals. Good at small sizes on screen and in print. Six weights with matching italics.

Gentium Plus

Designer: SIL International

Source: software.sil.org/gentium

Install: `apt install fonts-sil-gentium-plus`

Excellent multilingual coverage, particularly for African and Asian languages using extended Latin. Includes a full set of diacritical characters and IPA symbols. The preferred choice for linguistic texts, Bible typesetting, and any document requiring extensive Unicode Latin coverage.

Charis SIL

Designer: SIL International

Source: software.sil.org/charis

Install: `apt install fonts-sil-charis`

A warm, legible text face with thorough Unicode coverage. Designed to be readable at small sizes in print. Similar scope to Gentium but with a rounder, slightly more informal character.

Alegreya

Designer: Juan Pablo del Peral (Huerta Tipográfica)

Source: Google Fonts

A dynamic, literary text face with strong calligraphic influences. Works well for books and literary publications. Includes Alegreya Sans for pairing. Old-style figures and small capitals in the full version.

Cormorant

Designer: Christian Thalmann

Source: github.com/CatharsisFonts/Cormorant

A display-oriented Garamond revival in six stylistic sub-families (Cormorant Garamond, Infant, SC, Unicase, Upright Italic). Best used at

larger sizes for headings or poetry; less suited for extended body text at small sizes.

Sans-serif typefaces

Fira Sans

Designer: Carrois Apostrophe for Mozilla

Source: github.com/mozilla/Fira

LaTeX package: `fira`

Install: `apt install fonts-firacode fonts-font-awesome`

Designed for legibility on screen. A humanist sans with a large x-height and open apertures. Pairs naturally with EB Garamond or Libertinus. Nine weights with matching italics. Fira Sans Condensed available for tight spaces.

Inter

Designer: Rasmus Andersson

Source: github.com/rsms/inter, rsms.me/inter

Designed for user interfaces and screen use. A variable font with fine-grained weight and optical size control. Widely used in documentation and technical publications. Integrates very well with font-feature-settings in CSS for `tnum` (tabular numbers), `ss01` (disambiguation), and other features.

Source Sans

Designer: Paul D. Hunt (Adobe)

Source: github.com/adobe-fonts/source-sans

Package: Available via Google Fonts as “Source Sans 3”

Adobe’s open-source sans-serif, designed to complement Source Serif and Source Code Pro. Clean and functional. Extensive weight range. An excellent choice for headings when Source Serif is used for body text.

Lato

Designer: Łukasz Dziedzic

Source: Google Fonts

LaTeX package: `lato`

A warm humanist sans with semi-rounded details. Among the most widely used free sans-serif typefaces. Highly legible at small sizes. Good for documentation and technical reports.

Monospace typefaces

JetBrains Mono

Designer: JetBrains

Source: github.com/JetBrains/JetBrainsMono

Install: `apt install fonts-jetbrains-mono`

Designed specifically for software development. Increased height of lowercase letters, distinctive letterforms for ambiguous characters (o vs O, r vs l vs I), ligatures for programming tokens (`->`, `=>`, `!=`, `//`). Multiple weights. The recommended monospace font for code blocks throughout this book.

Source Code Pro

Designer: Paul D. Hunt (Adobe)

Source: github.com/adobe-fonts/source-code-pro

LaTeX package: sourcecodepro

Install: `apt install fonts-adobe-sourcecodepro`

Part of the Source type family. Clean, legible monospace with good disambiguation between similar characters. Multiple weights. No programming ligatures (which some prefer).

Fira Code

Designer: Nikita Prokopov

Source: github.com/tonsky/FiraCode

Adds programming ligatures to Fira Mono. The same base as Fira Sans Mono but with an extensive set of ligatures. If you want programming ligatures, JetBrains Mono and Fira Code are the two strongest options.

Iosevka

Designer: Belleve Invis

Source: github.com/be5invis/Iosevka

A highly customisable monospace. The build system allows specifying exact letterform variants for each character. Extremely narrow, suitable for very wide code. Many pre-built variants available for download without compiling.

Inconsolata

Designer: Raph Levien

Source: Google Fonts

LaTeX package: inconsolata

Inspired by Consolas (the Windows monospace standard). Clean, slightly condensed. Good for inline code and terminal output in documents.

Mathematical fonts

Computer Modern and Latin Modern

Designer: Donald Knuth (CM); GUST e-foundry (LM)

Source: Included in TeX Live; ctan.org/pkg/lm

LaTeX: Default fonts; `\usepackage{lmodern}` for Latin Modern

The default fonts of LaTeX. Computer Modern was designed by Knuth specifically for TeX. Latin Modern extends Computer Modern with additional character coverage. Reliable but visually associated with “typical LaTeX output.” Use when no other requirement governs the choice.

TeX Gyre collection

Designer: GUST e-foundry

Source: ctan.org/pkg/tex-gyre

LaTeX: Packages `tgpagella`, `tgtermes`, `tgschola`, etc.

Free alternatives to proprietary fonts: Pagella (Palatino), Termes (Times), Bonum (Bookman), Schola (Century Schoolbook), Heros (Helvetica), Cursor (Courier), Chorus (ITC Zapf Chancery), Adventor (Avant Garde). All include corresponding math fonts via the TeX Gyre Math packages.

STIX Two

Designer: STI Pub Companies

Source: github.com/stipub/stixfonts

LaTeX: stix2 package

A Times-based typeface with comprehensive Unicode mathematics coverage. Designed for scientific and technical publishing. The full Unicode math coverage makes it the most complete free mathematical font.

Libertinus Math

Designer: Caleb Maclennan

Source: github.com/alerque/libertinus

LaTeX: libertinus-otf package with unicode-math

The Unicode mathematics companion to Libertinus Serif. Recommended when Libertinus Serif is used for body text in XeLaTeX or LuaLaTeX documents with mathematics.

CJK fonts

Noto CJK

Designer: Google and Adobe

Source: github.com/notofonts/noto-cjk

Install: `apt install fonts-noto-cjk`

The most comprehensive free CJK font family. Covers Chinese Simplified, Traditional, Japanese, and Korean in both serif and sans-serif variants. Designed to complement the full Noto family for multilingual documents. Names: Noto Serif CJK SC/TC/JP/KR, Noto Sans CJK SC/TC/JP/KR.

Source Han

Designer: Adobe

Source: github.com/adobe-fonts/source-han-serif

The same fonts as Noto CJK under the Adobe brand name. Source Han Serif and Source Han Sans. Identical quality to Noto CJK.

Finding and installing fonts

Google Fonts (fonts.google.com): The largest collection of OFL fonts, all available for download. The web interface allows filtering by category, language, and number styles. Download the full family as a ZIP file containing all weights and styles.

Font Squirrel (fontquirrel.com): Curated collection of free commercial-use fonts with a strong filter for quality. Also hosts the WebFont Generator for converting fonts to WOFF2.

The League of Moveable Type (theleagueofmoveabletype.com): Small collection of high-quality OFL display and text fonts.

CTAN (ctan.org): TeX font packages including TeX Gyre, Latin Modern, and companions for most common typefaces. All fonts from CTAN are available through `tlmgr install`.

Linux installation

```
# From package manager
apt install fonts-ebgaramond fonts-libertinus fonts-firacode

# Manual installation
mkdir -p ~/.local/share/fonts/FamilyName
cp FontFile.ttf ~/.local/share/fonts/FamilyName/
fc-cache -f
```

```
fc-list | grep FamilyName # verify
```

LaTeX installation

```
# Via tlmgr (TeX Live package manager)
tlmgr install ebaramond libertinus lato fira inconsolata

# Verify
kpsewhich EBGaramond-Regular.otf
```

Appendix E: Further Reading and Bibliography

The works listed here informed or directly influenced the content of this book. They are grouped by subject; within each group, the most essential starting points are marked with a dagger (†). The bibliography that follows contains full details in alphabetical order.

Typography and type design

† Bringhurst, R. (2012). *The Elements of Typographic Style* (4th ed.). Hartley & Marks.

The fundamental text on Western typography. Covers the history and classification of typefaces, the principles of spacing, the conventions of page layout, and the relationship between typography and language. Every serious practitioner of document production should own a copy and consult it regularly. The book itself is a specimen of fine typography.

Hochuli, J. (2008). *Detail in Typography*. Hyphen Press.

A short, precise study of the smallest typographic decisions: letter spacing, word spacing, line spacing, and the relationship between them. Written with the authority of a practicing typographer. Essential for understanding why microtypographic adjustments matter.

Tracy, W. (1986). *Letters of Credit: A View of Type Design*. David R. Godine.

A thoughtful account of what makes typefaces legible and why different designs suit different purposes. Technically rigorous but readable. Invaluable for making informed typeface choices.

Lupton, E. (2010). *Thinking with Type* (2nd ed.). Princeton Architectural Press.

An accessible, visually rich introduction to typography for designers. Less concerned with fine detail than Bringhurst or Hochuli, but excellent at conveying the visual logic of typographic decisions.

Spiekermann, E., & Ginger, E. M. (2002). *Stop Stealing Sheep & Find Out How Type Works* (2nd ed.). Adobe Press.

A concise and opinionated introduction to typography from the designer of Meta and the redesign of the Economist's typeface. Good on the pragmatics of choosing and using typefaces.

TeX and LaTeX

† Knuth, D. E. (1984). *The TeXbook*. Addison-Wesley.

The definitive reference for TeX. Knuth wrote both the program and the book describing it, and the book is as carefully crafted as the software. Essential reading for understanding how TeX actually works. The “Dangerous Bend” sections cover advanced usage; a reader working through the main text gains a thorough understanding of the TeX model.

† Lamport, L. (1994). *LaTeX: A Document Preparation System* (2nd ed.). Addison-Wesley.

The original LaTeX manual, by the system's creator. Still a clear and concise introduction to LaTeX. The third edition of *The LaTeX Companion* (below) covers the current state of the ecosystem more thoroughly, but Lamport's book is shorter and often clearer on fundamentals.

Mittelbach, F., Gossens, M., & contributors. (2023). *The LaTeX Companion* (3rd ed., 2 vols.). Addison-Wesley.

The comprehensive reference for LaTeX package usage. Over 1,000 pages covering virtually every aspect of the LaTeX ecosystem: fonts, tables, mathematics, graphics, bibliographies, internationalisation, and more. The 3rd edition was substantially updated to cover contemporary packages and LaTeX₃.

Oetiker, T., Partl, H., Hyna, I., & Schlegl, E. *The Not So Short Introduction to LaTeX2ε*.

The standard free introductory guide. Regularly updated. Available as PDF from CTAN: ctan.org/pkg/lshort. A good starting point before tackling *The LaTeX Companion*.

Tantau, T. (2024). *The TikZ and PGF Manual*.

The complete reference for TikZ, the LaTeX diagram package. Over 1,200 pages. Available from CTAN and via `texdoc pgf`. A model of technical documentation.

Pandoc and Markdown

MacFarlane, J. *Pandoc User's Guide*. pandoc.org/MANUAL.html.

The complete reference for Pandoc. Exhaustive and accurate. The best way to learn a Pandoc feature is to read its section in the manual.

Gruber, J. *Markdown*. daringfireball.net/projects/markdown.

The original Markdown specification and philosophy. Short and worth reading to understand what Markdown was and was not designed to do.

Beber, A., & Lamire, C. (2022). *The Quarto Guide*. quarto.org/docs/guide.

The official guide to Quarto. Covers reproducible research, computational documents, and multi-format publishing.

Document engineering and build systems

Mecklenburg, R. (2004). *Managing Projects with GNU Make* (3rd ed.). O'Reilly.

The thorough reference for GNU Make. Free online edition available from the GNU project. Covers pattern rules, automatic variables, and the features needed for complex document build systems.

Kernighan, B. W., & Plauger, P. J. (1976). *Software Tools*. Addison-Wesley.

Classic book on the Unix philosophy of composable tools. The mindset it describes applies directly to the CLI document production workflows in this book.

Kernighan, B. W., & Pike, R. (1984). *The Unix Programming Environment*. Prentice-Hall.

A clear and elegant account of the Unix philosophy and its tools. Still valuable despite its age. The principles of pipeline composition and small, focused programs are directly applicable.

Web typography and CSS

Cederholm, D., & Ethan Marcotte. *CSS Typography. A Book Apart*. A concise guide to typographic decisions in CSS. Part of the A Book Apart series on web design and development.

Mozilla Developer Network. *CSS Fonts*. developer.mozilla.org/docs/Web/CSS/CSS_fonts
The complete reference for CSS font properties, including font-feature-settings, font-variation-settings, and the full @font-face syntax.

W3C. *CSS Fonts Module Level 4*. [w3.org/TR/css-fonts](https://www.w3.org/TR/css-fonts/).

The formal specification for CSS font properties. More authoritative than any tutorial, and often clearer than tutorials on the exact behaviour of edge cases.

History of printing and typography

McMurtrie, D. C. (1943). *The Book: The Story of Printing and Bookmaking*. Oxford University Press.

A comprehensive history from clay tablets to twentieth-century printing technology. The chapters on incunabula and early type design are particularly valuable for understanding where typographic conventions come from.

McLean, R. (1997). *The Thames and Hudson Manual of Typography*. Thames & Hudson.

A practical and historical account of typography from hot metal to early digital. Good on the transition between technologies and what was gained and lost at each stage.

Loxley, S. (2006). *Type: The Secret History of Letters*. I.B. Tauris. An accessible popular history of type design with engaging character portraits of type designers and their influences.

Online resources

CTAN --- Comprehensive TeX Archive Network. ctan.org

The repository for all TeX and LaTeX packages. Every package mentioned in this book is available here. Use `texdoc packagename` to read the documentation for any installed package.

TUG --- TeX Users Group. tug.org

The user organisation for TeX. Publishes *TUGboat*, a journal that has been running since 1980. Conference proceedings, technical articles, and a community of advanced users.

TeX Stack Exchange. tex.stackexchange.com

The Q&A site for LaTeX and TeX questions. An enormous archive of solved problems. The most reliable source for specific LaTeX troubleshooting.

Pandoc Discussion. github.com/jgm/pandoc/discussions

The official forum for Pandoc questions and feature discussions.

The Typst Forum. typst.app/forum

The community forum for Typst, with templates, packages, and troubleshooting.

Google Fonts Knowledge. fonts.google.com/knowledge

A well-written set of guides on type terminology, pairing, and best practices. Produced by Elliot Jay Stocks and others.

Bibliography

Bringhurst, R. (2012). *The Elements of Typographic Style* (4th ed.). Hartley & Marks.

Duffner, G., & Pardo, O. EB Garamond. *GitHub*. github.com/octaviopardo/EBGaramond

Gruber, J. (2004). Markdown. daringfireball.net/projects/markdown

Hochuli, J. (2008). *Detail in Typography*. Hyphen Press.

Hosny, K. Amiri. *GitHub*. github.com/alif-type/amiri

Knuth, D. E. (1984). *The TeXbook*. Addison-Wesley.

Knuth, D. E. (1986). *TeX: The Program*. Addison-Wesley.

Lamport, L. (1994). *LaTeX: A Document Preparation System* (2nd ed.). Addison-Wesley.

Loxley, S. (2006). *Type: The Secret History of Letters*. I.B. Tauris.

Lupton, E. (2010). *Thinking with Type* (2nd ed.). Princeton Architectural Press.

MacFarlane, J. (2006–). Pandoc: a universal document converter. pandoc.org

Maclennan, C. Libertinus fonts. *GitHub*. github.com/alerque/libertinus

- McLean, R. (1997). *The Thames and Hudson Manual of Typography*. Thames & Hudson.
- McMurtrie, D. C. (1943). *The Book: The Story of Printing and Book-making*. Oxford University Press.
- Mecklenburg, R. (2004). *Managing Projects with GNU Make* (3rd ed.). O'Reilly.
- Mittelbach, F., Gossens, M., Braams, J., Carlisle, D., & Rowley, C. (2023). *The LaTeX Companion* (3rd ed.). Addison-Wesley.
- Mozilla Foundation. CSS Fonts. *Mozilla Developer Network*. developer.mozilla.org/docs/Web/CSS/CSS_fonts
- Oetiker, T., Partl, H., Hyna, I., & Schlegl, E. (2023). *The Not So Short Introduction to LaTeX2e*. ctan.org/pkg/lshort
- Posit PBC. (2022–). Quarto: an open-source scientific and technical publishing system. quarto.org
- Prokopov, N. Fira Code. *GitHub*. github.com/tonsky/FiraCode
- SIL International. Gentium Plus. software.sil.org/gentium
- SIL International. Charis SIL. software.sil.org/charis
- Spiekermann, E., & Ginger, E. M. (2002). *Stop Stealing Sheep & Find Out How Type Works* (2nd ed.). Adobe Press.
- STI Pub Companies. (2021). STIX Two fonts. *GitHub*. github.com/stipub/stixfonts
- Tantau, T. (2024). *The TikZ and PGF Manual*. ctan.org/pkg/pgf
- Tracy, W. (1986). *Letters of Credit: A View of Type Design*. David R. Godine.
- Typst GmbH. (2023–). Typst documentation. typst.app/docs
- W3C. (2021). CSS Fonts Module Level 4. w3.org/TR/css-fonts

