go-pretty-pdf

Write Markdown. Ship a book.

Built: 2026-08-12 19:48:49 UTC

Sections: 3

Tags:

Table of Contents

[1.0.0] go-pretty-pdf

[2.0.0] CLI Reference

[3.0.0] Changelog

go-pretty-pdf

Transform a directory of MDX files into a beautiful, print-ready PDF via headless Chrome.

Library + CLI. Use it as a composable Go library or as a standalone command-line tool.

Fast. A 3,000-document book becomes a 3,535-page print-ready PDF in ~23 seconds — and validating those 3,000 documents takes 142 ms. Measured, reproducible numbers in BENCHMARKS.md.

Install

CLI (binary)

```
go install github.com/sazardev/go-pretty-pdf/cmd/pretty-pdf@latest
```

Library

```
go get github.com/sazardev/go-pretty-pdf
```

Requirements

- **Go 1.26+**
- **Chrome or Chromium** — optional. If none is found on your system, `pretty-pdf` automatically downloads and caches a small headless-only Chrome build the first time you run it (like Playwright/Puppeteer do). Already have Chrome installed? It's used as-is, nothing is downloaded. Prefer to control this yourself? Pass `--chrome-path /path/to/chrome` or set `PRETTY_PDF_CHROME_PATH`. Auto-download currently covers linux/amd64, darwin/amd64, darwin/arm64, and windows/amd64 — on linux/arm64 (no official build exists yet) install Chromium via your package manager and point `--chrome-path` at it.

Quick start

CLI

```
# Scaffold a new book project (interactive wizard)
pretty-pdf init my-book

# Build a PDF
pretty-pdf build --source my-book --out my-book.pdf

# Watch for changes and rebuild
pretty-pdf watch --source my-book --out my-book.pdf

# Validate MDX files
pretty-pdf check --source my-book
```

```go
package main

import (
        "context"
        "log"

        prettypdf "github.com/sazardev/go-pretty-pdf"
)

func main() {
        pdf, err := prettypdf.New(
                prettypdf.WithSourceDir("./docs"),
                prettypdf.WithOutputFile("output.pdf"),
                prettypdf.WithTitle("My Documentation"),
                prettypdf.WithAuthor("Jane Doe"),
        )
        if err != nil {
                log.Fatal(err)
        }

        if err := pdf.Build(context.Background()); err != nil {
                log.Fatal(err)
        }
}
```

How it works

```
MD/MDX files → Parse frontmatter & markdown → Transpile components → Compose HTML → Render
```

1. **Parse** — goldmark parses `.md` / `.mdx` files with YAML frontmatter. Fenced code blocks (```` ```go ```` , ```` ```python ```` , ...) are syntax-highlighted via Chroma, using a style paired to each theme's tone (e.g. Dracula for the `dark` theme, the Gruvbox style for the `gruvbox` theme, GitHub's light style everywhere else)
2. **Transpile** — custom components (`<DeepDive>` , `<Warning>` , `<Axiom>`) become styled HTML
3. **Compose** — HTML assembled with embedded template + CSS + auto-generated Table of Contents
4. **Render** — headless Chrome prints to PDF with headers, footers, and PDF bookmarks, then an automatic quality audit checks the result for overflowing content, broken images, low-contrast text, near-empty output, dead links, duplicate ids, broken TOC entries, unloaded fonts, at-risk page breaks, and headings at risk of being clipped by the print engine (see `pretty-pdf build` 's `Warnings` output, or `render.RenderToPDFWithAudit` in the library API)

Documents are sorted by their `[X.Y.Z]` frontmatter ID, not filename.

A `.md` / `.mdx` file doesn't strictly need a `---` frontmatter block either: if one is missing entirely, `id` and `title` are generated automatically from the filename, the same convention `.txt` uses below (`02-getting-started.mdx` → id `[2.0.0]` , title "Getting Started"; no numeric prefix → the next free major version, so it never

HTML, everything — unlike `.txt`. A `---` block that *is* present but fails to parse as YAML is still a hard error, so a typo in real frontmatter doesn't silently get treated as "no frontmatter".

`.txt` files are also accepted for freeform writing with zero setup: they have no frontmatter, so `id` and `title` are generated automatically from the filename (`03-field-notes.txt` → id `[3.0.0]`, title "Field Notes"; no numeric prefix → the next free major version, so it never collides with an explicitly numbered doc). Content is treated as literal text — blank lines become paragraphs, single line breaks become `<br>` — and is always HTML-escaped, so `.txt` loses component/raw-HTML customization but in exchange never executes anything the author typed. Good for a quick note dropped into an otherwise structured `.mdx` book.

Trust model

MDX is parsed with raw HTML passthrough enabled, and custom components don't escape their inner content — this lets authors embed arbitrary HTML/CSS for rich documents, but it also means a `.md` or `.mdx` file can contain a `<script>` tag that will execute during rendering. By default, headless Chrome's network access is blocked while rendering (see `WithNetworkAccess`), so scripts can't exfiltrate data or fetch remote content — but they still run. **Only build PDFs from source files you trust.** See SECURITY.md for details.

Performance

`go-pretty-pdf` keeps the Go-side pipeline (parse + validate + compose) sub-second even on large books; headless Chrome's print step is what drives PDF wall time. Measured on an i5-13500 (WSL2) with Chromium 150:

Source docs	check	epub	PDF pages	PDF wall time	PDF throughput
100	30 ms	27 ms	121	0.57 s	211 pages/s
1,000	62 ms	88 ms	1,180	2.99 s	394 pages/s
3,000	142 ms	248 ms	3,535	23.02 s	154 pages/s

Full table, hardware, and a reproducible recipe: BENCHMARKS.md. To measure your own machine:

```
PRETTY_PDF_CHROME_PATH=/usr/bin/chromium go run ./scripts/benchmark --color=false
```

```
---
id: "[1.0.0]"
title: "Getting Started"
subtitle: "A simple introduction"
tags: [example, intro]
difficulty: "beginner"
status: complete
completeness: 100
depends_on: []
---

# Welcome to Your Book

This is the first chapter.

## Variables

You can use {{key}} syntax for variable substitution: running {{product}} v{{version}}.
```

Required frontmatter fields: `id` (format `[X.Y.Z]`), `title`.

Built-in components

Component	Usage	Appearance
`<DeepDive>`	`<DeepDive title="Details">...</DeepDive>`	Blue info panel
`<Warning>`	`<Warning title="Note">...</Warning>`	Orange warning panel
`<Axiom>`	`<Axiom>...</Axiom>`	Green italic quote

Register custom components via `WithComponent()`:

```go
prettypdf.WithComponent("Callout", func(attrs map[string]string, inner string) string {
        level := attrs["level"]
        return fmt.Sprintf(`<div class="callout callout-%s">%s</div>`, level, inner)
})
```

Configuration

Create a `go-pretty-pdf.yml` in your project:

```yaml
subtitle: "A Complete Guide"
author: "Jane Doe"
source: book
output: out.pdf
theme: default

css: custom.css
template: custom-template.html

vars:
  product: "go-pretty-pdf"
  version: "1.0"

lint:
  require_frontmatter: [id, title]
  require_id_format: "[X.Y.Z]"
  no_duplicate_ids: true
  max_heading_depth: 3

render:
  timeout: 60s
  paper: a4
  margin_top: 20mm
  margin_bottom: 20mm
  margin_left: 15mm
  margin_right: 15mm
  header_title: "{{title}}"
```

Library API

```go
// Constructor with functional options
pdf, err := prettypdf.New(opts...)

// All-in-one build pipeline
pdf.Build(ctx)

// Step-by-step pipeline
docs, _ := pdf.ParseDir()
errs := pdf.ValidateDoc(doc)
html, _ := pdf.ComposeHTML(docs)
pdf.Render(html)

// Quality audit from the most recent Build/Render call (nil if neither ran yet)
audit := pdf.LastAudit()

// Validation-only
errs, _ := pdf.Validate(ctx)

// Lower-level: render straight to PDF and get the audit report back
report, err := render.RenderToPDFWithAudit(html, "out.pdf", render.DefaultOptions())
```

Option	Description
`WithSourceDir(dir)`	MDX source directory (default: `book`)
`WithOutputFile(path)`	Output PDF path (default: `out.pdf`)
`WithTitle(title)`	Document title
`WithSubtitle(sub)`	Document subtitle
`WithAuthor(author)`	Document author
`WithCSS(css)`	Custom CSS content string
`WithTemplate(html)`	Custom HTML template string
`WithTheme(t)`	Apply a raw `theme.Theme` (no customization/section toggles)
`WithThemeName(name, opts)`	Resolve a theme by name (builtin, custom, or file path) with color/font/section customization
`WithComponent(name, handler)`	Register custom MDX component
`WithValidator(v)`	Custom validation logic
`WithTimeout(d)`	Chrome render timeout (default: 60s)
`WithHeaderTitle(t)`	PDF header title
`WithVerbose(bool)`	Enable verbose logging
`WithVars(map)`	Variable substitution map
`WithRenderMargins(t,b,l,r)`	PDF margins in inches
`WithPaperSize(w,h)`	Paper size in inches
`WithConfig(cfg)`	Apply source/output/title/subtitle/author from config
`WithConfigCSSAndTemplate(cfg)`	Load CSS/template from config file paths
`WithFullConfig(cfg)`	Apply the entire config struct (source, CSS/template, theme, vars, render settings) in one call
`WithNetworkAccess(bool)`	Allow headless Chrome to make network requests while rendering (default: `false`, blocked)

Themes

Seventeen built-in themes, each a palette/typography layer over one shared structural stylesheet — clean and professional by default, easy to customize without writing CSS, and extendable with your own custom themes:

`terminal` · `blueprint` · `ivy` · `government` · `resume` · `legal` · `latex` · `gruvbox`

```
# Pick a theme, tweak colors/fonts/density, drop sections you don't want
pretty-pdf build --theme corporate \
  --color-primary "#0ea5e9" --font-heading "Georgia, serif" \
  --no-cover --no-page-numbers --density compact

# Scaffold your own reusable theme
pretty-pdf theme new my-report --from corporate
pretty-pdf theme list
```

```
prettypdf.WithThemeName("corporate", theme.Options{
        Colors:   theme.Colors{Primary: "#0ea5e9"},
        Sections: theme.Sections{Cover: theme.BoolPtr(false)},
})
```

Custom themes live in `<name>.theme.yml` files (project-local `./themes/` or a global themes directory) and `extends` a builtin theme. Full reference, all customization fields, and the `pretty-pdf theme` command family: see docs/cli.md#themes.

CLI reference

```
pretty-pdf build     Build a PDF from MDX source files
pretty-pdf epub      Build an EPUB from MDX source files (no Chrome required)
pretty-pdf check     Validate MDX files without building
pretty-pdf theme     List, inspect, and manage themes
pretty-pdf init      Scaffold a new book project (interactive wizard)
pretty-pdf watch     Watch for changes and rebuild automatically
pretty-pdf serve     Preview MDX as HTML with live reload (no Chrome required)
pretty-pdf version   Print the version number
```

Run `pretty-pdf <command> --help` for the full flag list of any command.

Global flags: `--config`, `--source`, `--verbose`, `--no-color`, `--quiet`

License

MIT — see LICENSE.

CLI Reference

Overview

`pretty-pdf` transforms a directory of MDX files into a print-ready PDF and/or a reflowable EPUB 3 file. Documents are sorted by their `[X.Y.Z]` frontmatter ID, not by filename.

GitHub: https://github.com/sazardev/go-pretty-pdf

Requirements

- **Chrome or Chromium** — optional. If `pretty-pdf` can't find one on your system, it automatically downloads and caches an official, automation-only "chrome-headless-shell" build the first time you run a command that renders a PDF (`build` , `watch`). This mirrors what tools like Playwright/Puppeteer do; `serve` never needs Chrome since it only previews HTML. The download is cached under your OS's user cache directory (e.g. `~/.cache/go-pretty-pdf/chrome` on Linux) and reused on every later run.
  - Already have Chrome/Chromium installed? It's detected and used automatically — nothing is downloaded.
  - Want to pin a specific binary instead (skip detection/download entirely)? Pass `--chrome-path /path/to/chrome` or set the `PRETTY_PDF_CHROME_PATH` environment variable.
  - Supported for auto-download: linux/amd64, darwin/amd64, darwin/arm64, windows/amd64. On linux/arm64 (no official build exists yet), install Chromium via your system's package manager and use `--chrome-path` .
- Go 1.26+ (if building from source).

Usage

```
pretty-pdf [command] [flags]
```

Flag	Default	Description
`--config`	`""`	Path to config file
`--source`	`"book"`	Source MDX directory
`--chrome-path`	`$PRETTY_PDF_CHROME_PATH`	Path to a Chrome/Chromium executable (skips auto-detection/download)
`--verbose`	`false`	Verbose output
`--no-color`	`false`	Disable colored output
`--quiet`	`false`	Suppress non-error output
`-h, --help`		Help for any command

Commands

`build`

Parse MDX files, validate them, compose HTML, and render to PDF and/or EPUB.

```
pretty-pdf build [flags]
```

`--format`	`"pdf"`	Output formats: `pdf`, `epub`, or `pdf,epub`
`--out`	`"out.pdf"`	Output path (extension determines single format; base name for multi-format)
`--title`	`""`	Book title
`--subtitle`	`""`	Book subtitle
`--author`	`""`	Book author
`--theme`	`"default"`	Theme name (builtin, custom, or a `.theme.yml` / `.css` path) — see Themes
`--css`	`""`	Custom CSS file path (overrides the theme entirely)
`--template`	`""`	Custom HTML template file path (overrides the theme's HTML)
`--cover-image`	`""`	Custom cover image (`.png` / `.jpg` / `.jpeg` / `.svg` / `.webp`); the cover page is sized to the image's own dimensions, replacing the text cover
`--timeout`	`""`	Render timeout (e.g. `30s`, `1m`)
`--language`	`"en"`	EPUB language (BCP-47 tag, e.g. `en`, `es`)
`--json`	`false`	Output as JSON
`--no-cover`	`false`	Omit the cover page
`--no-toc`	`false`	Omit the table of contents
`--no-page-numbers`	`false`	Omit page numbers
`--no-header`	`false`	Omit the running page header
`--no-outline`	`false`	Skip PDF bookmarks/outline — faster on very large documents
`--no-tagged-pdf`	`false`	Skip PDF accessibility tagging (PDF/UA) — faster on very large documents
`--color-primary`	`""`	Theme override: primary color (e.g. `#1a56db`)
`--color-accent`	`""`	Theme override: accent color
`--color-text`	`""`	Theme override: body text color
`--color-muted`	`""`	Theme override: muted/caption text color
`--color-bg`	`""`	Theme override: page background color

`--font-heading`	`""`	Theme override: heading font family
`--font-body`	`""`	Theme override: body font family
`--font-code`	`""`	Theme override: code font family
`--density`	`""`	Spacing density: `compact`, `normal`, or `relaxed`
`--allow-network-fonts`	`false`	Allow fetching Google Fonts declared by the theme (enables network access)

Build Pipeline

The `build` command runs through these stages per format:

1. **Parse** — Read and parse all MDX files in the source directory
2. **Validate** — Check frontmatter, duplicate IDs, heading depth, content warnings
3. **PDF compose** — Assemble HTML with TOC, cover page, and embedded CSS/template (PDF only)
4. **PDF render** — Generate PDF via headless Chrome, then run an automatic quality audit (PDF only)
5. **EPUB write** — Package chapters directly into EPUB 3, no Chrome needed (EPUB only)

Chrome is only required when `pdf` is in the format list. An `epub`-only build (`--format epub`) skips Chrome detection entirely.

PDF Quality Audit

Right after rendering, `build` runs a best-effort audit of the composed document and reports anything worth a second look — it's advisory for DOM/layout findings and only reports at `error` severity for output that's actually corrupt. The final summary's `Warnings` count reflects this (and `--json`'s `warnings` array lists them in full). Checks:

`overflow-x`	Content wider than its box (long code lines, wide tables/images) that print will clip instead of wrap
`overflow-y`	Content taller than a fixed-height box, so it clips when printed
`broken-image`	An `<img>` that never resolved to real pixels
`image-low-res`	An image displayed at more than ~2x its intrinsic size, so it will look pixelated on paper
`empty-content`	The document has almost no visible text — usually a sign composition silently produced nothing
`low-contrast`	Text whose contrast ratio fails WCAG 2.2 (4.5:1 normal, 3:1 large text) against its effective background
`heading-clip-risk`	A heading that forces a page break without enough top margin to clear the print engine's header/margin strip, so its top would render clipped
`broken-anchor`	An `<a href="#fragment">` with no matching element — dead in-document links break TOC and PDF bookmarks
`duplicate-id`	The same `id` attribute used twice, which breaks anchors, the TOC, and PDF bookmarks
`toc-mismatch`	The TOC links to an id that doesn't exist, or a body section heading has no TOC entry
`font-load-fail`	A font family the page requests could not be loaded (missing local font, or a Google Font blocked by the default network lockdown) and will silently fall back
`page-break-inside-risk`	A table/code block without `page-break-inside: avoid`, so print can slice it mid-row
`line-break-risk`	A block with `orphans` / `widows` below 2, so a single line can be stranded at the top/bottom of a page
`page-count`	The generated PDF has no detectable pages — the output file may be empty or corrupt
`pdf-empty`	The generated PDF is zero bytes — the output file is empty
`pdf-eof-missing`	The generated PDF is missing its `%%EOF` marker — the output may be truncated or corrupt
`unused-component`	A component registered via `WithComponent()` was never used in any document — check the tag spelling

The audit reads the composed HTML before it's handed to the print engine, so it can't see two things that live purely inside Chrome's own print pipeline: the fixed ~0.2in header/footer inset and the actual page-break slicing (both covered by `base.css`'s own layout rules instead — see the CHANGELOG for the bugs those rules exist to prevent).

Pre-flight Checks

Before the pipeline starts, `build` verifies (per selected formats):

- Chrome/Chromium is available (only when `pdf` is in the format list)

- At least one `.md` / `.mdx` / `.txt` file is present
- Each output path's directory is writable
- Custom CSS file exists (if specified)
- Custom template file exists (if specified)
- Custom cover image exists and is a supported format (`.png` / `.jpg` / `.jpeg` / `.svg` / `.webp` , if specified)

`check`

Parse and validate all MDX files without building a PDF. Previously named `validate` .

```
pretty-pdf check [flags]
```

Flag	Default	Description
`--strict`	`false`	Treat content warnings as errors

`epub`

Parse MDX files, validate them, and write a single EPUB 3 file — no Chrome/Chromium involved, unlike `build` . Each MDX document becomes its own chapter, in the same order as the PDF's table of contents.

```
pretty-pdf epub [flags]
```

`--out`	`"out.epub"`	Output EPUB path
`--title`	`""`	Book title
`--subtitle`	`""`	Book subtitle (used as the EPUB's `dc:description`)
`--author`	`""`	Book author
`--theme`	`"default"`	Theme name (theme CSS is converted to reflowable EPUB form)
`--css`	`""`	Custom CSS file path (overrides theme entirely)
`--cover-image`	`""`	Custom cover image (`.png` / `.jpg` / `.jpeg` / `.svg` / `.webp`), full-bleed as the first page
`--language`	`"en"`	Book language (BCP-47 tag, e.g. `en`, `es`)
`--color-primary`	`""`	Theme override: primary color
`--color-accent`	`""`	Theme override: accent color
`--color-text`	`""`	Theme override: body text color
`--color-muted`	`""`	Theme override: muted/caption color
`--color-bg`	`""`	Theme override: page background color
`--font-heading`	`""`	Theme override: heading font family
`--font-body`	`""`	Theme override: body font family
`--font-code`	`""`	Theme override: code font family
`--density`	`""`	Spacing density: `compact`, `normal`, or `relaxed`
`--allow-network-fonts`	`false`	Allow fetching Google Fonts declared by the theme

Reuses `--source` / `--config` like every other command, and `render.cover_image` from `go-pretty-pdf.yml` if `--cover-image` isn't passed — the same cover image works for both `build` and `epub`. Unlike PDF output — which uses `@page` rules and print-oriented layout — EPUB uses the same theme system through `ResolveForEPUB`, which produces a reflowable stylesheet (relative units, no print-only rules, no cover/TOC/page-number sections) that works across e-reader devices.

`theme`

List, inspect, and manage themes.

```
pretty-pdf theme show <name>
pretty-pdf theme new <name> [flags]
pretty-pdf theme add <path> [flags]
```

`theme list`

Prints every builtin theme (name + description) followed by any custom themes discovered in `./themes/` (project) and the global themes directory (`~/.config/pretty-pdf/themes` on Linux, via `os.UserConfigDir()`).

`theme show <name>`

Resolves a theme (builtin, custom, or a `.theme.yml` / `.css` path) with no customization and prints its final, fully-assembled CSS to stdout — useful to inspect a theme or pipe it somewhere (`pretty-pdf theme show dark > dark.css`).

`theme new <name>`

Scaffolds a starter `<name>.theme.yml` you can hand-edit.

Flag	Default	Description
`--from`	`"default"`	Builtin theme to base the scaffold on
`--global`	`false`	Write to the global themes directory instead of `./themes`

Refuses to overwrite an existing file.

`theme add <path>`

Imports an existing `.theme.yml` or raw `.css` file as a managed custom theme (a loose `.css` file is wrapped into a minimal `.theme.yml` with `extends: default` and the file's content as its `css:` block).

Flag	Default	Description
`--as`	`""`	Name to register the imported theme under (default: derived from the file name)
`--global`	`false`	Copy to the global themes directory instead of `./themes`

`init`

Scaffold a new book project with sample MDX files and configuration.

```
pretty-pdf init [directory] [flags]
```

Interactive mode (default): runs a terminal form asking for title, author, theme, source directory.

`--bare`	`false`	Non-interactive init with flags
`--title`	`"My Book"`	Book title (for `--bare`)
`--author`	`"go-pretty-pdf"`	Book author (for `--bare`)
`--theme`	`"default"`	Book theme (for `--bare`)
`--json`	`false`	Output as JSON

`serve`

Parse MDX files, compose HTML, and serve with live reload on file changes. No Chrome required.

```
pretty-pdf serve [flags]
```

Flag	Default	Description
`--port`	`8080`	HTTP server port

Uses Server-Sent Events for live reload. Watches `.md`, `.mdx`, `.txt`, `.yaml`, and `.yml` files for changes.

`watch`

Watch the source directory for changes and rebuild the PDF on every file change.

```
pretty-pdf watch [flags]
```

Debounces changes by 300ms. Watches `.md`, `.mdx`, `.txt`, `.yaml`, and `.yml` files. Prints a build/error summary on `Ctrl+C`.

`version`

Print the version number.

```
pretty-pdf version
```

`completion`

Generate shell completion scripts.

```
pretty-pdf completion [bash|zsh|fish|powershell]
```

bash	`pretty-pdf completion bash > /etc/bash_completion.d/pretty-pdf`
zsh	`pretty-pdf completion zsh > "${fpath[1]}/_pretty-pdf"`
fish	`pretty-pdf completion fish > ~/.config/fish/completions/pretty-pdf.fish`
powershell	`pretty-pdf completion powershell > _pretty-pdf.ps1` then `. .\_pretty-pdf.ps1`

Config File

`go-pretty-pdf.yml` is auto-discovered by walking up from the working directory. Can also be specified explicitly with `--config`.

```yaml
title: "My Book"
subtitle: "A journey into MDX-powered PDFs"
author: "Jane Doe"
source: book
output: out.pdf
theme: corporate
css: custom.css
template: custom.html
vars:
  version: "1.0"
  year: "2026"

theme_options:
  colors:
    primary: "#1a56db"
    accent: "#0ea5e9"
  fonts:
    heading: "Georgia, serif"
    google_fonts: ["Inter:400,600"]   # only fetched with allow_network_fonts: true
  sections:
    cover: true
    toc: true
    page_numbers: true
    header: true
  density: normal        # compact | normal | relaxed
  allow_network_fonts: false

lint:
  require_frontmatter:
    - id
    - title
  no_duplicate_ids: true
  max_heading_depth: 3

render:
  timeout: 30s
  paper: A4            # or: letter, legal, 6x9in, 152.4mm x 228.6mm
  margin_top: 20mm
  margin_bottom: 15mm
  margin_left: 15mm
  margin_right: 15mm
  header_title: "My Book"
```

Field	Default	Description
`title`	`"Document"`	Book title
`subtitle`	`""`	Book subtitle
`author`	`"go-pretty-pdf"`	Book author
`source`	`"book"`	Source MDX directory
`output`	`"out.pdf"`	Output path (extension-added per format when using `--format` from CLI)
`theme`	`""`	Theme name (builtin, custom, or a `.theme.yml` / `.css` path) — see Themes
`css`	`""`	Path to custom CSS file (overrides the theme entirely)
`template`	`""`	Path to custom HTML template file (overrides the theme's HTML)
`vars`	`{}`	Template variables for `{{key}}` substitution
`theme_options`	`{}`	Theme customization — see Themes

`lint` fields

Field	Default	Description
`require_frontmatter`	`["id", "title"]`	Required frontmatter fields
`no_duplicate_ids`	`true`	Reject duplicate document IDs
`max_heading_depth`	`5`	Maximum allowed heading depth

Field	Default	Description
`timeout`	`""`	Chrome render timeout (e.g. `30s`, `1m`)
`paper`	`""`	Paper size: `letter`, `legal`, `A4`, custom dimensions (`6x9in`, `152.4mm x 228.6mm`, `6x9`), or empty for CSS default
`margin_top`	`""`	Top margin as CSS unit (`20mm`, `1in`, `10mm`, `2cm`, `12pt`, `96px`)
`margin_bottom`	`""`	Bottom margin as CSS unit
`margin_left`	`""`	Left margin as CSS unit
`margin_right`	`""`	Right margin as CSS unit
`header_title`	`""`	Header title in rendered PDF
`cover_image`	`""`	Path to a custom cover image (`.png` / `.jpg` / `.jpeg` / `.svg` / `.webp`), or `--cover-image`

For a full-bleed page (a dark theme's background reaching every edge, no white border), set all four margins to `0mm` / `0in` and disable the header and page numbers (`theme_options.sections.header` / `page_numbers: false`, or `--no-header --no-page-numbers`) — Chrome reserves a small fixed strip for the header/footer that can't otherwise be removed.

Custom cover image

Setting `render.cover_image` (or `--cover-image`) replaces the theme's text cover with a full-bleed page built from that image alone — no title, subtitle, or theme styling on it. Unlike every other page, which uses `render.paper`, this page is sized to the image's own pixel dimensions exactly (at 96px/in): a square image gets a square cover page, a portrait photo gets a portrait-shaped page matching its aspect ratio precisely. The rest of the document (TOC, sections, page numbers) keeps the configured paper size untouched. It always wins over `theme_options.sections.cover` and any theme's own cover markup, regardless of which is set first.

Supported formats: `.png`, `.jpg`, `.jpeg`, `.svg` (dimensions from `width` / `height` attributes or `viewBox`), `.webp`.

```
render:
  cover_image: assets/cover.png
```

Themes

Seventeen built-in themes are available, each a palette/typography layer over a shared structural stylesheet (`theme/assets/base.css`):

`default`	professional	Clean, professional look that fits any technical document.
`minimal`	minimal	Stripped down: smaller type, no borders, maximum simplicity.
`modern`	professional	Sans-serif with generous whitespace and bold accent underlines.
`classic`	editorial	Serif, traditional book layout — ink on paper.
`corporate`	professional	Structured blue/gray palette for client-facing reports.
`dark`	dark	Dark background with light text. Best for on-screen PDFs.
`academic`	academic	Formal serif layout for theses, papers, and reports.
`editorial`	editorial	Magazine-style display headings and pull-quote blockquotes.
`sepia`	warm	Warm, sepia-toned palette for long, comfortable reading sessions.
`terminal`	technical	All-monospace, terminal-inspired look for technical references.
`blueprint`	technical	Dark technical blueprint palette with monospace type and cyan highlights.
`ivy`	institutional	Classic Ivy League university letterhead: forest green and gold on cream.
`government`	institutional	Formal official-document palette: navy and bronze, centered headings.
`resume`	resume	Clean, ATS-friendly sans-serif for CVs and one-pagers — no cover or TOC.
`legal`	formal	Stark, formal brief style: black ink, no color as decoration.
`latex`	academic	Mathematical/scientific paper look with automatic section numbering.
`gruvbox`	technical	Retro warm dark palette inspired by the popular Gruvbox editor theme.

Run `pretty-pdf theme list` to see this list plus any custom themes, and `pretty-pdf theme show <name>` to print a theme's final resolved CSS.

Customizing a theme without writing CSS

`theme_options` (config) or the matching `--color-*` / `--font-*` / `--density` / `--no-*` flags (CLI) customize any theme — builtin or custom — without touching CSS:

```
pretty-pdf build --theme corporate \
  --color-primary "#0ea5e9" --font-heading "Georgia, serif" \
  --no-cover --no-page-numbers --density compact
```

`colors.primary/accent/text/muted/background`	CSS custom properties for the theme's palette
`fonts.heading/body/code`	Font-family overrides (system-safe stacks recommended)
`fonts.google_fonts`	Google Fonts family names (e.g. `["Inter:400,600"]`) — only fetched when `allow_network_fonts: true`, since network access is otherwise blocked during rendering
`sections.cover/toc/page_numbers/header`	`true` / `false` /unset (unset = theme's own default)
`density`	`compact`, `normal`, or `relaxed` — adjusts line-height and a handful of spacing rules
`allow_network_fonts`	Enables outbound network access during rendering so `fonts.google_fonts` can be fetched

Section toggles set via `--no-cover` / `--no-toc` / `--no-page-numbers` / `--no-header` only apply to the default HTML template; a custom `--template` owns its own HTML and must implement any toggles itself (the default template gates its cover block on `{{if .ShowCover}}`).

Custom themes

A custom theme is a `<name>.theme.yml` file that extends a builtin theme:

```yaml
name: my-report
description: "Client report with a teal accent"
extends: corporate

colors:
  accent: "#0d9488"
fonts:
  heading: "Georgia, serif"
sections:
  page_numbers: false
density: normal

css: |
  /* raw CSS appended last — wins over everything above */
  .cover h1 { text-transform: uppercase; }
```

Custom themes are discovered by name in `./themes/` (project-local, checked first) and then in the global themes directory (`~/.config/pretty-pdf/themes` on Linux). Use them the same way as a builtin: `--theme my-report` or `theme: my-report` in config.

Manage them with:

```
pretty-pdf theme add ./some-theme.theme.yml        # import an existing theme file
pretty-pdf theme add ./some.css --as my-report     # or wrap a plain CSS file
pretty-pdf theme list                              # see builtins + everything discovered
pretty-pdf theme show my-report                    # print the fully resolved CSS
```

A `--theme` value ending in `.theme.yml` / `.css` is treated as a direct file path instead of a name, so you can also point straight at a file without installing it into a themes directory.

Template Variables

Available in HTML templates:

Variable	Description
`{{.Title}}`	Book title
`{{.Subtitle}}`	Book subtitle
`{{.Author}}`	Book author
`{{.CSS}}`	Inline CSS string
`{{.Body}}`	Composed document body
`{{.BuiltAt}}`	Build timestamp
`{{.TotalDocs}}`	Number of documents
`{{.Keywords}}`	Tags from documents

Environment

- `NO_COLOR` environment variable is respected (disables colored output).
- `PRETTY_PDF_CHROME_PATH` sets the default for `--chrome-path`: a specific Chrome/Chromium executable to use, skipping auto-detection and auto-download.

Exit Codes

Code	Meaning
`0`	Success
`1`	General error (parsing, validation, rendering, config)

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

[Unreleased]

0.11.0 - 2026-08-12

Added

- **More exports from `scripts/docsgen`**: one EPUB per builtin theme (`go-pretty-pdf-docs-<theme>.epub`) plus a canonical default, raw markdown sources (`README.md` , `docs.md` , `CHANGELOG.md`), a client/agent search index (`docs-search.json`), a machine-readable build manifest (`version.json`), a plain-text `sitemap.txt` , and a full performance report (`report.json`).
- **Aggressive performance reporting in `docsgen`**: the summary now shows wall time, artifact count/size, throughput (artifacts/s and MiB/s), phase-sum parallelism ratio, per-group artifact tables with mean/p95/min/max/stddev, and an ASCII-bar phase timeline. `--bench` additionally re-renders the theme PDFs at `jobs=1` to headline the real parallel speedup, and `_site/report.json` persists every metric in machine-readable form.
- **Ten new quality-audit checks**, expanding the automatic PDF audit from 6 rules to 16:
  - `overflow-y` — content taller than a fixed-height box, clipped on print.
  - `image-low-res` — an image rendered at more than ~2x its intrinsic width, so it will look pixelated on paper.
  - `broken-anchor` — an `<a href="#fragment">` with no matching element (dead in-document links break the TOC and PDF bookmarks).
  - `duplicate-id` — the same `id` attribute used twice.
  - `toc-mismatch` — a TOC entry with no target id, or a body section with no TOC entry.
  - `font-load-fail` — a requested font family the browser can't resolve (missing local font, or a Google Font blocked by the default network lockdown).
  - `page-break-inside-risk` — a table/code block without `page-break-inside: avoid` , which print can slice mid-row.
  - `line-break-risk` — a block with `orphans` / `widows` below 2, which can strand a single line at the top/bottom of a page.
  - `unused-component` — a component registered via `WithComponent()` that no document used (a typo'd tag is the usual cause).
  - `pdf-eof-missing` — the finished PDF lacks its `%%EOF` marker (truncated/corrupt output).

"the PDF is genuinely corrupt" from advisory layout warnings (`pdf-empty` , `pdf-eof-missing` , `page-count`).

- **WCAG 2.2 contrast thresholds** in `low-contrast` : 4.5:1 for normal text and 3:1 for large text (≥18.66px, or ≥14px bold), replacing the old flat 2.2:1 cutoff that only caught obviously unreadable pairs.
- **`mdx.Parser.ComponentUsage()` / `ComponentNames()` / `ResetComponentUsage()`** so callers can inspect which custom components a parse actually exercised, and **`render` PDF-byte auditing** now tolerates Chrome's trailing-newline `%%EOF` .
- **`render.NewBrowser` , `render.RenderToPDFWithAuditBrowser` , and `prettypdf.WithSharedBrowser`** : boot one headless Chrome allocator (with startup-tuned flags) and render many documents against it instead of launching a fresh Chrome per `Build` . Useful for keeping a persistent browser across a batch of renders; measured on this machine it does not beat running each render concurrently, which remains the main parallelism lever. The new browser setup also became the default for `RenderToPDFWithAuditContext` , so every render picks up the reduced-startup flag set.
- **Large-document audit sampling**: the DOM audit's style-reading checks (`overflow` , `low-contrast` , `page-break-inside-risk` , `line-break-risk` , `heading-clip-risk` , TOC coverage, font-load) now sample the first few thousand matching elements instead of walking the entire tree, since layout reads (`getComputedStyle` , `scrollWidth/Height`) dominate on documents with tens of thousands of nodes and the findings are deduped/capped anyway. Documents render far faster at scale (measured: 5,000-document PDF went from >120s timeout to ~60s).
- **Speed flags for very large documents**: `--no-outline` (`prettypdf.WithGenerateDocumentOutline(false)`) and `--no-tagged-pdf` (`prettypdf.WithGenerateTaggedPDF(false)`) disable Chrome's post-print outline/bookmark build and accessibility tagging, the two most expensive post-print steps. Measured on a ~500-page document: both off is ~30% faster than both on (~1.1s vs ~1.6s); on docsgen's full theme-PDF set they cut wall time ~10%. Defaults keep both on — bookmarks and PDF/UA accessibility are features.
- **Full CLI help overhaul**: every command's `--help` now explains what it does, exit status where relevant, performance guidance for large books, and realistic example commands; `theme --help` lists all 17 built-in theme names.
- **`_headers` deploy config**: the generated site now ships a `_headers` file (per-path `Cache-Control` + security headers incl. CSP) that Cloudflare Pages and Netlify honor out of the box.

Changed

- The audit report's `unused-component` finding is attached to `PDF.LastAudit()` after `Build` , so the same report carries both visual and authoring signals.
- `version` now respects `--no-color` and prints a single `v` prefix (it previously rendered the default `v0.10.0` as `vv0.10.0`).
- The docs page shares the landing page's appbar (brand, links, theme dropdown, GitHub button, mobile hamburger) and its sidebar, hero, and content styling were aligned with the landing page's visual language; the docs default theme is now `default` .

- The `duplicate-id` message used the raw DOM node in its text; it now names both elements (`#dup` on … (also on `#dup`)).
- The new `%%EOF` check originally flagged Chrome's well-formed output because Chrome appends a newline after the marker; the check now tolerates trailing whitespace.
- `docsgen` 's `report.json` could panic with `index out of range` when computing the fastest/slowest artifact names (indices into the ok-durations slice were confused with indices into the results slice); the scan now tracks names directly.
- **The docs site's theme switcher showed `classic` 's palette for `default` / `minimal` / `modern`** : the generated `:root` fallback rode on `classic` , whose block sits mid-list, so its unconditional `:root` rule (same specificity as the `[data-site-theme]` blocks, later in source order) overrode the earlier palettes. The `:root` fallback now rides on `default` , the first theme in `theme.List()` order.
- **CLI-reference fidelity**: corrected the documented flag name `--formats` (the flag is `--format`), the `lint.max_heading_depth` default (5, not 3), and added the undocumented `pdf-empty` audit check to the audit table.

0.10.0 - 2026-07-16

Added

- **Multi-format `build` (`--format pdf,epub`)**: render PDF and EPUB in a single pass from one source. A base `--out` name (no extension) is given both extensions automatically, and an `out.pdf` / `out.epub` value pins the matching format while deriving the other. New library option `prettypdf.WithFormats(...)` and the standalone `pretty-pdf epub` command share the same pipeline.
- **SVG and WebP cover images** for `--cover-image` / `render.cover_image` , alongside the existing PNG/JPG/JPEG formats.
- **Custom paper dimensions** in `render.paper` (and the matching CLI plumbing): `6x9in` , `6x9` , `6in x 9in` , `152.4mm x 228.6mm` — exact print-on-demand trim sizes beyond the named `letter` / `legal` / `A4` .

0.9.0 - 2026-07-15

Added

- `PDF.Warnings()` : returns the non-fatal configuration warnings recorded by `New` (an unresolvable theme name, an unreadable `--css` / `--template` file) — each of these already printed to stderr, but had no way to be detected programmatically without scraping output.

Fixed

- **A checksum-mismatched Chrome download was left behind on Windows instead of being removed**: `downloadFile` 's "remove the partial/corrupt file on error" cleanup was registered as a `defer` *before* the "close the file handle" `defer` , so it ran first (defers execute LIFO) and tried to delete the destination file while it was

for a later step to unwittingly pick up. The defers are now registered in the correct order.

- **`New()` swallowed configuration warnings unless `--verbose` / `WithVerbose(true)` was set**: a typo'd theme name or an unreadable `--css` / `--template` file silently fell back to the previous value with zero indication anything was wrong, since the warning was only ever printed to stderr behind the verbose flag. Warnings are now always printed; see `PDF.Warnings()` above for programmatic access. `New()`'s `error` return is unchanged — an unresolved theme is still non-fatal by design.
- **`{{var}}` substitution was non-deterministic and could chain into other variables**: `substituteVars` replaced one variable at a time by looping over the vars map (Go's map iteration order is randomized) and doing a sequential `ReplaceAll` on the *already-substituted* text, so a variable whose value itself contained another variable's `{{placeholder}}` could get expanded a second time — the same input could render differently across runs. Substitution is now a single `strings.Replacer` pass over the original text, so every placeholder expands exactly once, deterministically. `Parser.vars` is also now mutex-guarded for concurrent `SetVars` / `ParseFile` use.
- **Nested components of the same type left a stray closing tag**: `<Warning><Warning>...</Warning></Warning>` only matched the innermost pair (the transpile regex is intentionally non-greedy), leaving a literal `</Warning>` in the rendered output. Same-name nesting is now unwound innermost-first by repeating the replacement until no match remains. Component tags also now accept single-quoted attribute values and attributes other than `title` (e.g. `<Warning id="x" title='Heads up'>`), instead of only the exact literal `title="..."` shape.
- **The native header/footer strip could pick up colors from document body text, not just the real theme**: `pageChrome` extracted `--pdf-*` CSS variables from the *entire* composed document (cover, TOC, every chapter), and that parser matches the pattern wherever it appears — so a chapter merely discussing or showing that CSS syntax (e.g. a code sample) could silently override the theme's actual background/muted-text color used to paint the header/footer. Vars are now extracted from the document's `<style>` block only.
- **A custom theme's raw CSS (or a `--css` file) could break out of the `<style>` block**: unlike the color/font override values `theme.Resolve` already sanitizes, a `.theme.yml`'s `css:` escape hatch and `--css` / `--template` file content were inserted into `<style>{{.CSS}}</style>` completely unescaped (by design, so legitimate CSS survives intact) — including the literal sequence `</style>`, which would end the style element early and let arbitrary markup follow. That sequence is now neutralized before insertion; nothing else in the CSS is touched.
- **A failed EPUB rebuild could destroy a previously good `.epub` file**: `epub.Write` opened `outputPath` directly with `os.Create`, truncating it immediately, before anything had actually been written successfully — a failure partway through (a bad chapter, a full disk, an interrupted process) left a corrupt/incomplete archive in its place. `Write` now builds the archive in a temp file and renames it into place only on success, matching the same pattern already used for cover+body PDF merging.
- **Canceling a `context.Context` mid-`Build` / `Validate` had no effect until the render stage**: `Build` never checked `ctx` between parsing, validation, and composing, and `Validate` accepted a `ctx` parameter but never referenced it at all. Both now check `ctx.Err()` between stages (and per-document during validation) and stop early once canceled.

and its decoded pixel dimensions passed straight through as `PrintToPDF`'s exact paper size, so an unusually large file — or one whose header merely claims huge dimensions — could trigger an oversized in-memory read and an oversized print request. Now capped at 32MB and 20000px per side.

- **Auto-downloaded Chrome builds silently skipped their integrity check when GCS omitted the MD5 header**: `chromemgr.downloadFile` verifies the download against the `X-Goog-Hash` GCS reports, but when that header was absent it fell through with no message at all, even though the file is about to be `chmod +x`'d and executed. Now surfaced as a warning via the existing progress callback.
- EPUB's `renderCoverXHTML` / `renderNavXHTML` template errors are now wrapped with context, matching `renderChapterXHTML`.

0.8.0 - 2026-07-15

Added

- **Custom cover image** (`--cover-image` / `render.cover_image`): a PDF cover built from a `.png` / `.jpg` / `.jpeg` file, full-bleed and sized to exactly the image's own pixel dimensions (a square image gets a square cover page) instead of the document's configured paper size — the rest of the document is unaffected. `Page.printToPDF` only accepts one paper size per call, so the cover is printed as its own single-page PDF and stitched in front of the normally-rendered body via a raw page merge (new dependency: `github.com/pdfcpu/pdfcpu`, which preserves each side's own page size rather than forcing a uniform one). Replaces the theme's text cover outright, regardless of `theme_options.sections.cover` or option/config ordering. New library API: `prettypdf.WithCoverImage`, `render.Options.CoverImagePath`, `config.RenderConfig.CoverImage`.
- **`epub` command**: converts the same MDX source into a single EPUB 3 file — no headless Chrome involved at all, unlike `build`. Each document becomes its own XHTML chapter in reading order; the frontmatter `[X.Y.Z]` ID hierarchy that drives the PDF's table of contents now also drives a real nested `nav.xhtml` outline (plus a flattened `toc.ncx` for older readers). `--cover-image` / `render.cover_image` is shared with `build`: the same image becomes a standalone full-bleed cover page. New dependency-free package `github.com/sazardev/go-pretty-pdf/epub` (`epub.Write`), building the archive with only the standard library (`archive/zip`, `crypto/rand` for the required `dc:identifier` UUID). goldmark's renderer now emits XHTML-style self-closing void elements (`<img/>`, `<hr/>`, `<br/>`) globally — still valid HTML5, so the PDF pipeline is unaffected, but required for EPUB's chapter files to be well-formed XHTML.

Fixed

- **`build` silently failed on large books with `chromedp render: page load error net::ERR_ABORTED`**: rendering navigated Chrome to a `data:text/html;charset=utf-8;base64,...` URI holding the entire composed document, which Chrome (via chromedp/CDP's `Page.navigate`) aborts once the encoded payload crosses roughly 2MB — with no message indicating size was the cause. A book with a few hundred thousand words of prose and code crosses that threshold easily. Rendering now writes the composed HTML to a temporary file and navigates to it via a `file://` URL instead (removed once the page is captured), which has no such ceiling; the existing default-network-blocking behavior (`NetworkAccess: false` still blocks external

`TestRenderToPDFLargeDocumentPastOldDataURILimit` generates a document past the old limit and confirms it renders.

- **`context.Context` cancellation was ignored during rendering**: `PDF.Build(ctx)` never propagated `ctx` into the Chrome render — canceling it (client disconnect, SIGINT wired to context cancellation) had no effect and the render ran to completion or `Options.Timeout` regardless. New `render.RenderToPDFWithAuditContext(ctx, ...)` roots the Chrome allocator in the caller's context; `Build` now uses it. `render.RenderToPDF` / `RenderToPDFWithAudit` keep their exact previous signatures (rooted in `context.Background()`) for API stability.
- **Data race on live-reload's SSE channel** (`pretty-pdf serve`): an `/events` connection read `reloadCh` with no lock while a rebuild concurrently closed and reassigned it, which could make an in-flight connection observe a stale, already-closed channel and never see a subsequent reload. Also hardened `notifyReload` itself to close-and-replace under a single lock instead of a separate read-then-write pair, which could otherwise panic with "close of closed channel" under concurrent calls.
- **Data race on watch-mode build stats** (`pretty-pdf watch`): `WatchStats` was mutated from the rebuild loop and read from the Ctrl+C signal handler with no synchronization. Now guarded by a mutex.
- **A failed `--cover-image` merge could destroy a previously good PDF**: `mergeCoverAndBody` truncated the output file immediately via `os.Create` , before the pdfcpu merge had actually succeeded. A merge failure partway through (malformed cover PDF, pdfcpu error) left an empty/partial file in place of the last good output. Now written to a temp file and renamed into place only on success.

Security

- **HTML injection via the `id` frontmatter field**: `mdx.AnchorID` passed the `id` field straight through into `id` / `href` HTML attributes with no escaping (`compose.ComposeHTML` / `buildTOC` bypassed `html/template` 's autoescaping by building markup with `fmt.Fprintf` + `template.HTML`). A crafted `id` like `1.0.0"><script>...` could inject arbitrary markup into the composed document. `AnchorID` now sanitizes to a safe HTML-id charset at the source, plus defense-in-depth escaping at both call sites.
- **CSS injection via theme color/font overrides**: `theme.Resolve` interpolated `Options.Colors` / `Options.Fonts` values (and Google Fonts family names) straight into generated CSS/ `url(...)` with no escaping, letting a value containing `;` / `{` / `}` / `'` / `)` break out of its declaration and inject arbitrary rules (e.g. re-enabling a hidden `.cover`). Values are now sanitized before use.
- **Malformed/malicious EPUB chapter content**: goldmark's raw-HTML passthrough (`WithUnsafe`) means an MDX author's literal `<br>` or ` ` / `&mdash;` reaches `epub.Write` unmodified — valid enough for Chrome's lenient HTML parser (the PDF path), but not well-formed XML, breaking strict EPUB readers. Chapter bodies are now reparsed/re-serialized through `x/net/html` before templating, which always self-closes void elements and resolves entities to literal text.
- **Auto-downloaded Chrome binary had no integrity check**: `chromemgr.EnsureChrome` 's one-time download fetched `chrome-headless-shell` over HTTPS with no verification before `chmod +x` and executing it. The Chrome for Testing manifest publishes no signed checksum, but the binaries are served from Google Cloud

transport corruption and a class of tampering proxy), and a mismatched/corrupt download is removed rather than left on disk.

- `golang.org/x/image` **bumped to v0.43.0**: fixes two known panics decoding a malformed/large WEBP image (GO-2026-5061, GO-2026-4961), reachable via `--cover-image`'s dimension probing (`render.coverImageDimensionsIn`).
- `golang.org/x/net` **bumped to v0.55.0**: the new `epub.xhtmlifyFragment` (above) calls `x/net/html`'s `ParseFragment` / `Render`, which at v0.54.0 carried five known issues (GO-2026-5025/5027/5028/5029/5030 — a DoS parsing arbitrary HTML, an XSS via duplicate attributes, and related HTML-parsing correctness bugs) in the exact code path `xhtmlifyFragment` exercises on every chapter's body.

0.7.0 - 2026-07-07

Added

- `gruvbox` **builtin theme** (17 total): retro warm dark palette inspired by the popular Gruvbox editor theme (`#282828` background, orange `#fe8019` accent, monospace throughout), requested by name.
- `theme.ExtractCSSVars(css)`: shared parser for `--pdf-*` custom property declarations, extracted out of `scripts/docsgen` so `render` can reuse it too.
- **Automatic PDF quality audit**: `build` now runs a best-effort visual/structural audit right after rendering and reports what it finds — advisory only, it never fails the build. Checks: `overflow-x` (content wider than its box that print will clip instead of wrap), `broken-image`, `empty-content` (near-zero visible text, usually a sign composition silently produced nothing), `low-contrast` (visible text too close in color to its effective background), `heading-clip-risk` (a heading that forces a page break without enough top margin to clear the print engine's header strip — the general form of the bug fixed below, now caught automatically for custom themes/CSS too), and `page-count` (the generated PDF has no detectable pages). Surfaced as a real `Warnings` count and itemized list in `build`'s terminal output and `--json`'s `warnings` array. New library API: `render.RenderToPDFWithAudit` (returns an `*render.AuditReport` alongside the existing error; `render.RenderToPDF`'s signature is unchanged) and `PDF.LastAudit()`.

Fixed

- **Dark themes no longer print with a white border**: `page.PrintToPDF`'s margin area sits outside the page's paintable content box, so Chrome never fills it with the document's own background — every dark theme (`dark`, `blueprint`, `gruvbox`, ...) rendered as a dark rectangle floating inside a plain white page. Left/right margins are now 0 (the reading margin moved to CSS `padding` on `<body>` instead, which the theme's background paints straight through for true edge-to-edge bleed), and the native top/bottom margin — kept only because that's the one place the running header/page-number footer can render — is now painted with the theme's own `--pdf-bg` / `--pdf-muted` instead of hardcoded white/gray, so it blends into the page instead of showing up as a separate band. `displayHeaderFooter` is now only enabled when a header or page numbers are actually wanted, so documents with both disabled get a fully clean, gap-free page.

version honors an `@page` margin (even `margin: 0`) over `Page.printToPDF`'s imperative `marginTop` / `marginBottom` / `marginLeft` / `marginRight`, so `render.Options` and `go-pretty-pdf.yml`'s `render.margin_*` had no visible effect whensoever they disagreed with `@page` — silently, with no error. `@page` no longer declares a margin at all; the imperative API parameters are now the single real source of truth.

- **`margin_top` / `margin_bottom` / `margin_left` / `margin_right: "0mm"` in `go-pretty-pdf.yml` was silently ignored**: `WithFullConfig` detected "was a margin configured?" from the *parsed* value being non-zero, making an explicit `0mm` indistinguishable from the field being absent — a config asking for a true full-bleed page got the default margins instead. Now gated on the config *string* being non-empty.
- **Chapter titles and the "Table of Contents" heading rendered clipped, overlapping the running header**: every `h1` that starts a fresh page (`page-break-before: always`, or the TOC's own heading right after the forced break following the cover) had `margin-top: 0`, putting its text flush against the top of the page's content box. With a header/page-number footer enabled, chrome-headless-shell clips roughly the first 0.3in of whatever sits there — confirmed by disabling the header (clean render) and by testing plain body text landing on a fresh page through natural pagination instead of a forced break (also clean), so the defect is specific to content flush against a *forced* page break while a header/footer is displayed. `h1` now keeps a `0.35in` top margin, comfortably clearing the dead zone on every theme (none of which override it), guarded by `TestBaseCSSH1HasTopMarginBuffer`.

Known limitation: Chrome reserves a small, fixed ~0.2in strip at the very top/bottom of the page whenever a header/footer is displayed at all, regardless of the configured margin or the header/footer template's own CSS — confirmed by direct testing, not something this project's CSS can override. It's colored to match the theme so it no longer looks like a stray white band, but on a dark theme with a header or page numbers enabled you may still see a hairline. Disable both (`--no-header --no-page-numbers`, or `theme_options.sections.header` / `page_numbers: false`) and set all four margins to `0mm` / `0in` for a page with zero gap on every side.

0.6.0 - 2026-07-07

Added

- **8 new builtin themes**, bringing the total to 16 — each required only a CSS file and a registry entry in `theme/builtin.go` to appear correctly in the CLI (`pretty-pdf theme list`) and the docs website (switcher + its own dogfooded PDF), with zero other files touched:
  - `sepia` — warm, soft palette for long reading sessions
  - `terminal` — all-monospace, terminal-inspired, with a `$` prompt-style cover
  - `blueprint` — dark technical palette with cyan highlights and a dashed cover border
  - `ivy` — classic Ivy League university letterhead (forest green and gold)
  - `government` — formal official-document palette (navy and bronze, centered headings, double rules)
  - `resume` — clean ATS-friendly sans-serif for CVs; disables cover/TOC/page numbers/header by default
  - `legal` — stark, formal brief style with no color used as decoration

2., ...) via CSS counters

- Five new theme categories: `warm`, `technical`, `institutional`, `resume`, `formal`.
- `theme.Theme.Accented`: marks builtin themes (classic, modern, corporate, editorial, terminal, blueprint, ivy, government) that use their accent color as a bold structural element (cover border, accent blockquote) rather than just for links.
- `resumeSections`: a `ResolvedSections` preset (all sections off) for themes meant for short, single-flow documents.

Changed

- **docs site theme automation**: `scripts/docsgen` now derives every theme's colors, fonts, and accent treatment straight from `theme.List()` and each theme's own CSS at build time (see `themevars.go`), instead of a hand-maintained, hardcoded copy of the palette in `site.css`. Adding a builtin theme to `theme/builtin.go` is now the only step needed for it to appear in the docs site's theme switcher, get correct swatch colors, and get its own dogfooded "docs as a PDF" download — nothing to keep in sync by hand. This is also what caused the "incoherent blue" bug fixed earlier: the site's copy had drifted from the real theme CSS.

0.5.0 - 2026-07-07

Added

- **Automatic Chrome management** (`chromemgr` package): `pretty-pdf build` / `watch` no longer require Chrome/Chromium to be installed manually. If none is found, a small official "chrome-headless-shell" build (Google's Chrome for Testing distribution) is downloaded and cached under the OS user cache dir on first use, then reused on every later run — no different from what Playwright/Puppeteer do. A system-installed Chrome/Chromium is always preferred and used as-is when present. New `--chrome-path` flag / `PRETTY_PDF_CHROME_PATH` env var let users pin a specific binary and skip detection entirely. Covers linux/amd64, darwin/amd64, darwin/arm64, windows/amd64 (linux/arm64 has no upstream prebuilt binary yet and falls back to a clear error asking for a manual install).
- **`render.Options.ChromeExecPath`** and **`prettypdf.WithChromeExecPath`**: point rendering at a specific Chrome/Chromium binary instead of chromedp's default discovery.
- **Named theme constants**: `NameDefault`, `NameMinimal`, `NameModern`, `NameClassic`, `NameCorporate`, `NameDark`, `NameAcademic`, `NameEditorial` exported from `theme` package — custom theme code and tests can now reference themes by constant instead of raw strings, eliminating goconst lint warnings across the codebase

Changed

- **Chrome startup timeout**: raised `chromedp.WSURLReadTimeout` to 45s in `RenderToPDF` and 15s in `CheckChromeAvailable`, plus boosted `CheckChromeAvailable` context timeout from 10s to 20s — prevents spurious "websocket url timeout reached" failures on cold/loaded CI runners

- **goconst lint**: all magic string literals for builtin theme names and categories replaced with named constants in `theme/builtin.go`; tests updated to use constants and share a `testCustomThemeName` const

0.4.0 - 2026-07-06

Added

- **Theme engine overhaul**: `theme` package now has a proper engine with `Resolve()`, `ResolveByName()` — merges base CSS + theme CSS + `:root` custom property overrides (colors, fonts, density) + section toggles
- **8 builtin themes**: default, minimal, modern, classic, corporate, dark, academic, editorial — each with dedicated CSS files embedded via `//go:embed`, with categories (professional, editorial, dark, academic, minimal)
- **Custom theme system**: `<name>.theme.yml` files extending a builtin theme, discovered in `./themes/` (project-local) and `~/.config/pretty-pdf/themes` (global) — full YAML schema with `extends`, `colors`, `fonts`, `sections`, `density`, and raw `css` escape hatch
- **`theme` CLI subcommand** with 4 subcommands:
  - `theme list` — shows builtin + custom themes with descriptions
  - `theme show <name>` — prints fully-resolved CSS to stdout
  - `theme new <name>` — scaffolds a starter `.theme.yml` (with `--from` and `--global` flags)
  - `theme add <path>` — imports existing `.theme.yml` or `.css` files as managed custom themes (with `--as` and `--global` flags)
- **Section toggles** (cover, TOC, page numbers, header) controlled via:
  - CLI flags: `--no-cover`, `--no-toc`, `--no-page-numbers`, `--no-header`
  - Config: `theme_options.sections.cover`, `.toc`, `.page_numbers`, `.header` (nullable booleans — unset = theme default)
  - Template gating: `{{if .ShowCover}}` / `{{if .ShowTOC}}` wrapping the cover block and TOC in `template.html`
  - CSS gating: `.cover{display:none !important;}` / `.toc{display:none !important;}` appended by `Resolve()` for disabled sections
  - `render.Options.PageNumbers` and `render.Options.ShowHeader` — when disabled, Chrome header/footer templates render `<div></div>` (empty)
- **Color/font customization**: `--color-*` and `--font-*` CLI flags + `theme_options.colors` / `fonts` in config — drives `--pdf-*` CSS custom properties in a `:root` block
- **Density control**: `--density compact|normal|relaxed` CLI flag + `theme_options.density` — adjusts `--pdf-line-height` and `--pdf-space-scale`
- **Google Fonts support**: `fonts.google_fonts` in theme YAML/config, fetched only when `allow_network_fonts: true` (network disabled by default for security)
- **`WithThemeName(name, opts)` option**: resolves a theme by name (builtin, custom, or file path) with full opts customization, wiring section toggles into `composeOpts` / `renderOpts`

requests

- **Config struct**: `ThemeOptionsConfig`, `ColorsConfig`, `FontsConfig`, `SectionsConfig` with full YAML serialization
- **Test suite**: 20 new tests across `theme/`, `pdf_test.go`, `compose/compose_test.go`, `render/render_test.go`, `config/config_test.go`

Changed

- `build` and `theme` CLI commands now carry expanded help text with full `Long` descriptions, `Example` blocks, and dynamic theme name listing
- `README.md` theme section updated: lists all 8 builtin themes, documents `WithThemeName(name, opts)`, adds CLI usage examples for theme customization
- `theme.Theme` struct: now includes `Description`, `Category`, `Sections` (resolved defaults), and `CSS` comes from dedicated asset files instead of raw Go strings
- `WithTheme(t)` — now applies CSS only (no template); section toggles must be set separately
- `WithConfigCSSAndTemplate` — resolves `cfg.Theme` via `ResolveByName` with full `ThemeOptionsConfig` customization before applying explicit CSS/template file overrides
- Old hardcoded `theme.Minimal.CSS` inline string replaced by `//go:embed assets/minimal.css`
- `cmd/pretty-pdf/main.go` — `--theme` flag usage now lists all builtin theme names dynamically from `theme.List()`
- `cmd/pretty-pdf/config.go` — maps CLI flags to `cfg.ThemeOptions` (colors, fonts, sections, density, network)
- `docs/cli.md` — comprehensive docs for all new flags, config fields, theme subcommands, custom theme workflow

Removed

- Inline `minimalCSS` string in `theme/theme.go` — CSS now lives in `theme/assets/*.css`

Security

- Google Fonts (`fonts.google_fonts`) require explicit `allow_network_fonts: true` — network access remains blocked by default during headless Chrome rendering

0.3.0 - 2026-07-06

Added

- `WithFullConfig(cfg)` option: applies the entire `config.Config` struct (source, output, title, subtitle, author, CSS/template, theme, vars, render settings) in a single call
- `WithNetworkAccess(bool)` option: control whether headless Chrome can make outbound network requests during rendering (default: `false`)

`cmd/pretty-pdf/config.go`)

- `config.PaperLetter` constant for YAML config comparisons
- `render.Options.NetworkAccess` field, with default network blocking via `chromedp/cdproto/network`
- Concurrent-safe `ComponentRegistry` (`sync.RWMutex` protecting the handler map)
- `headerTitleSet` tracking in `PDF`: prevents `New()` from overwriting an explicit header title with the document title
- Deferred warning buffer in `PDF`: `WithConfigCSSAndTemplate` file-read failures are collected and flushed by `New()` after all options run, making warning output order-independent from `WithVerbose`
- Comprehensive test suite: `pdf_test.go` (16 tests), `compose/compose_test.go` (6 tests), `compose/toc_test.go` (3 tests), `config/units_test.go` (2 tests), `config/units_test.go` (2 tests), `mdx/component_test.go` (1 race test), `mdx/parser_test.go` (7 tests), `render/render_test.go` (4 tests)
- Showcase book: 8-document MDX example under `examples/showcase/` with 8 custom components (`Callout`, `Badge`, `Steps`, `Card`, `Stat`, `Timeline`, `Quote`, `Progress`)
- Showcase integration test: `examples/showcase_test.go` verifies compose output and full PDF rendering
- Trust model documentation in `README.md`, `SECURITY.md`, and package-level `doc.go`

Changed

- `cmd/pretty-pdf/buildOpts` simplified to `WithFullConfig` + `WithValidator` (removed duplicated config-to-option mapping)
- `WithConfigCSSAndTemplate` file-read warnings now deferred to `New()` instead of printed inline (order-independent from `WithVerbose`)

Security

- **Network access blocked by default** during headless Chrome rendering: prevents SSRF/exfiltration from untrusted MDX content via `<script>`, `<img>`, `<link>`, etc.
- Detailed trust model documented across `README.md`, `SECURITY.md`, and `doc.go`

0.2.1 - 2026-07-03

Fixed

- **Data URI corruption**: switched from `url.QueryEscape` to base64 encoding for the HTML data URI passed to Chrome — `QueryEscape` converts spaces to `+`, which Chrome does not decode in data URIs, so every space in rendered text showed up as a literal `+`

0.2.0 - 2026-07-03

Added

- `--bare` flag on `init`: minimal non-interactive project scaffolding
- `--port` flag on `serve`: configure HTTP server port
- `ValidateAll(docs)` method on `PDF` and `Validator` interface for batch validation
- `TestAnchorID` test covering bracket-stripping behavior
- `.component-warning-title` CSS class for `<Warning>` title styling
- Template placeholders (`{{BOOK_TITLE}}` , `{{AUTHOR_NAME}}` , `{{SOURCE_DIR}}` , `{{THEME}}`) in init scaffold files
- Support for h4 and h5 heading levels in MDX documents, with matching TOC styling and CSS

Changed

- `AnchorID` uses `strings.Trim` instead of `strings.ReplaceAll` for bracket removal
- `PrintValidationSummary` now shows per-file breakdown (passed/errored/warned)
- `render.DefaultOptions()` drives margin defaults in config instead of hardcoded 0.8
- `FindConfig` resolves from `os.Getwd()` instead of relative `.`
- Warning component uses `html.EscapeString` from stdlib instead of custom `escapeHTML`

Removed

- `AnchorIDRaw` function (unused)
- `PrintSummary` method from `PipelineProgress` (unused)
- Deprecated `LintConfig` fields: `RequireIDFormat` , `RequireLowercaseFilenames` , `CheckBrokenLinks`
- Version banner from `init` , `check` , and `watch` commands (noise reduction)
- Custom `dirname` helper in render — replaced by `filepath.Dir`
- Per-doc validation loop in `build.go` — replaced by single `ValidateAll` call

Fixed

- `url.PathEscape` → `url.QueryEscape` for correct data URI encoding in Chrome rendering

0.1.0 - 2026-07-02

Added

- MDX parser based on goldmark with YAML frontmatter support
- Custom component transpiler: `<DeepDive>` , `<Warning>` , `<Axiom>` built-in
- Variable substitution (`{{key}}`) before parsing
- HTML composition with embedded template and print CSS
- Table of Contents auto-generated from `[X.Y.Z]` IDs
- Headless Chrome PDF rendering via chromedp

- CLI with cobra: `build`, `check`, `init`, `watch`, `version` commands
- Animated pipeline UI with lipgloss styles, spinner, and progress panels
- Interactive init wizard using huh form library
- File watcher with 300ms debounce for live rebuilds
- YAML config file (`go-pretty-pdf.yml`) with auto-discovery
- Configurable lint validator: required frontmatter, ID format, duplicates, heading depth
- Paper size presets: A4, Letter, Legal
- CSS unit parser: mm, cm, in, pt, px
- Comprehensive example suite with custom components, themes, and CSS
- 18 functional options on the root PDF type
- Step-by-step API: ParseDir, ValidateDoc, ComposeHTML, Render
- Partial parsing: per-file errors collected, valid docs proceed
- GitHub Actions CI (lint, test, vet, build on 3 OS) and release pipeline (goreleaser)
- Local Makefile with lint, test, build, and release-dry-run targets